USENIX Association

Proceedings of the
2023 USENIX Annual Technical Conference

July 10–12, 2023
Boston, MA, USA

ISBN 978-1-939133-35-9

Conference Organizers

Program Co-Chairs

Julia Lawall, *Inria*

Dan Williams, *Virginia Tech*

Program Committee

Reto Achermann, *University of British Columbia*

Godmar Back, *Virginia Tech*

Saurabh Bagchi, *Purdue University*

Jia-Ju Bai, *Tsinghua University*

Yungang Bao, *Institute of Computing Technology, Chinese Academy of Sciences*

Yaniv Ben-Itzhak, *VMware Research*

Annette Bieniusa, *Technische Universität Kaiserslautern*

Roberto Bifulco, *NEC Laboratories Europe*

Laurent Bindschaedler, *Max Planck Institute for Software Systems (MPI-SWS)*

Eleanor Birrell, *Pomona College*

William Bolosky, *Microsoft*

Philippe Bonnet, *IT University of Copenhagen*

Sara Bouchenak, *INSA Lyon*

Nathan Bronson, *Rockset*

Maria Carpen-Amarie, *Huawei Zurich Research Center*

Somali Chaterji, *Purdue University*

Lydia Chen, *Delft University of Technology*

Yu Chen, *Tsinghua University*

Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*

David Cock, *ETH Zurich*

Dave Dice, *Oracle*

Thaleia Dimitra Doudali, *IMDEA Software Institute*

Abhinav Duggal, *Dell EMC*

Eric Eide, *University of Utah*

Dan Feng, *Huazhong University of Science and Technology*

Xinwei (Mason) Fu, *Amazon Web Services*

Wei Gao, *University of Pittsburgh*

Jana Giceva, *Technische Universität Munich*

Kartik Gopalan, *Binghamton University*

Redha Gouicem, *Technische Universität Munich*

Xiaohui (Helen) Gu, *North Carolina State University*

Nastaran Hajinazar, *Intel Labs*

Kyle Hale, *Illinois Institute of Technology*

Niranjan Hasabnis, *Intel Labs*

Chris Hawblitzel, *Microsoft Research*

Michio Honda, *University of Edinburgh*

Liting Hu, *Virginia Tech*

Yu Hua, *Huazhong University of Science and Technology*

Călin Iorgulescu, *Oracle Labs*

Zsolt István, *Technische Universität Darmstadt*

Anand Iyer, *Microsoft Research*

Hani Jamjoom, *IBM T. J. Watson Research*

Yu Jiang, *Tsinghua University*

Myoungsoo Jung, *Korea Advanced Institute of Science and Technology (KAIST)*

Asim Kadav, *Tonal*

Vasiliki Kalavri, *Boston University*

Anuj Kalia, *Microsoft*

Sudarsun Kannan, *Rutgers University*

Sanidhya Kashyap, *EPFL*

Wook-Hee Kim, *Konkuk University*

Ricardo Koller, *Google*

Kenji Kono, *Keio University*

Youngjin Kwon, *Korea Advanced Institute of Science and Technology (KAIST)*

Sándor Laki, *Eötvös Loránd University*

Michael Le, *IBM T. J. Watson Research*

Eunji Lee, *Soongsil University*

Baptiste Lepers, *Université de Neuchâtel*

Alberto Lerner, *University of Fribourg*

Yu Liang, *City University of Hong Kong*

Jean-Pierre Lozi, *Inria*

Youyou Lu, *Tsinghua University*

Xiaosong Ma, *Qatar Computing Research Institute, Hamad Bin Khalifa University*

Sarah Meiklejohn, *University College London and Google*

Mike Mesnier, *Intel Labs*

Subrata Mitra, *Adobe Research*

Apoorve Mohan, *IBM T. J. Watson Research Center*

Amy L. Murphy, *Bruno Kessler Foundation*

Ruslan Nikolaev, *The Pennsylvania State University*

Pierre Olivier, *The University of Manchester*

Amy Ousterhout, *University of California, San Diego*

Yuvraj Patel, *University of Edinburgh*

Fernando Pedone, *University of Lugano*

Kevin Pedretti, *Sandia National Laboratories*

Jan Rellermeyer, *Leibniz University Hannover*

Larry Rudolph, *Two Sigma Investments, LP*

Leonid Ryzhyk, *VMware Research*

Russell Sears, *Crystal DB*

Mohammad Shahrad, *University of British Columbia*

Yizhou Shan, *Huawei Cloud*

Liuba Shrira, *Brandeis University*

Georgios Smaragdakis, *Delft University of Technology*

Nik Sultana, *Illinois Institute of Technology*

Cheng Tan, *Northeastern University*

Vasily Tarasov, *IBM Research - Almaden*

Alain Tchana, *ENSIMAG*

Daniel R. Thomas, *University of Strathclyde*

Gaël Thomas, *Télécom SudParis - Institut Polytechnique de Paris*

Animesh Trivedi, *Vrije Universiteit Amsterdam*

Theodore Ts'o, *Google*

Chia-Che Tsai, *Texas A&M University*

Shay Vargaftik, *VMware*

Lluís Vilanova, *Imperial College London*

Chen Wang, *IBM T. J. Watson Research Center*

Jason Waterman, *Vassar College*

Emmett Witchel, *The University of Texas at Austin and Katana Graph*

Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*

Suzhen Wu, *Xiamen University*

Suli Yang, *NetApp*

Eiko Yoneki, *University of Cambridge*

Feng Zhang, *Tsinghua University*

Jie Zhang, *Peking University*

Yongle Zhang, *Purdue University*

Yuan Zhang, *Fudan University*

Zheng Zhang, *Rutgers University*

Yibo Zhu, *ByteDance Inc.*

External Reviewers

Message from the USENIX ATC '23 Program Co-Chairs

Introduction

Welcome to the 2023 USENIX Annual Technical Conference (USENIX ATC '23). We are excited to be holding an in-person event with minimal visa issues requiring remote participation. Similar to last year, USENIX ATC '23 is co-located with OSDI. We very much look forward to meeting everyone in the systems community whether they attend USENIX ATC, OSDI, or both. The rest of this document provides some insights into the submission and selection process that culminated in 65 accepted works that will be presented at the conference.

Submissions

As in previous years, USENIX ATC '23 solicited three types of papers. In addition to full length, 11-page research papers, authors could submit 5-page short research papers that describe complete and properly evaluated ideas using fewer pages. Finally, to align with the USENIX mission of bringing together researchers in academia and systems practitioners, we continued the practice of soliciting papers describing the design, implementation, analysis, and experience with real-world deployment of systems and networks in a deployed systems track. These "Deployed Systems" papers had different criteria for acceptance from research papers, not needing to present new ideas or results to be accepted, but needing to convey practical insights.

A submission to USENIX ATC '23 involved more than a single PDF file. On the HotCRP submission system, each submission also contained an artifact description that included further details about the experimental environment. Optionally, authors could include a textual description of changes from previous submissions to help in the case where reviewers may have seen prior iterations of the paper while on other PCs. Finally, if on the deployed systems track, authors were required to specify a justification for why the paper belonged in that track.

Program Committee Selection Process

We assembled a program committee with many goals in mind: good coverage across diverse computer-systems topics, balance between academia and industry, a mix of veterans of prior USENIX ATC PCs with individuals in early stages of their professional careers, geographic diversity, and adherence to the USENIX diversity and inclusion principles. The assembled PC had 107 members from 19 countries, including 44% from North America, 33% from EMEA and 21% from APAC. 70% of the PC was from academia and 30% from industry, though some PC members from academia were also affiliated with industry. 55% of our PC were veteran PC members who had served USENIX ATC at least once in the past 4 years. Our PC had 26% female representation, which is higher than recent years (e.g., 18% for USENIX ATC '22). The main areas of expertise of PC members were Storage (24%), Distributed Systems (25%), Operating Systems (26%), Security (15%), Networking (12%), and Machine Learning (17%).

For the PC selection process, which was done well in advance of the submission deadline, we drew from a pool of experienced PC members who had served at least once in the prior 4 iterations of USENIX ATC, removing those who had served 4 times in a row and those who were concurrently serving for OSDI. In making our invitations, we prioritized several factors, including the following: reviewers flagged in HotCRP as producing good reviews for previous conferences; recommendations of researchers (usually early career) from invitees who were unable to serve; topic matches, trying to anticipate the need for Machine Learning expertise based on topic ratios from last year; and female representation, as it has been low in the USENIX ATC community.

During the review process, there were a few cases in which we needed expertise for a paper in which all reviewers identified low levels of expertise. For these, we solicited recommendations from the PC and invited external reviewers.

On January 9th, 2023, we held a synchronous online PC pre-review meeting to go over the unique aspects of USENIX ATC '23 submissions, and an overview of the duties and processes involving PC members, including bidding, reviewing rounds, online discussions, the author rebuttal, the PC meeting, and shepherding. Although attended by both veteran and new PC members, we hoped the meeting helped to welcome new PC members and provide opportunities for questions about the process. We held the meeting twice (7 hours apart) in an attempt to accommodate the various timezones of our international PC.

Review Process

USENIX ATC '23 received 353 submissions across all tracks, which was 10% fewer than USENIX ATC '22. Of these, 22 (6%) were deployed systems papers and 19 (5%) were short papers. The most popular topics for submissions, as specified by

authors were: Clouds, clusters, data centers (29%); ML/AI (24%); Storage, file systems (23%); Parallel and Distributed Systems (22%); Operating Systems, Kernels (14%); and Networking (14%).

We adopted a double-blind review process to minimize bias with strict anonymity rules. Four papers were ultimately rejected due to including author names, directly identifying or sharing a name or title with an existing technical report, or directly linking to a github repository under the author's name or institution. We identified one of these cases prior to reviewing, but the others were detected during the reviewing process.

We rejected three other submissions without review due to violations of the formatting guidelines, two papers for exceeding the length limits, and one that was too short and did not contain sufficient detail. Of particular note was the misuse of appendices, including very long appendices and appendices that contain information that is integral to the paper.

In order to increase the quality and relevance of reviews, we ran a bidding process in which PC members had 8 days to bid on which papers they felt were in their competency/expertise area and for which they could provide knowledgeable reviews. PC members also updated their topic preferences in HotCRP, which along with bid values were used by the HotCRP algorithm to assign papers. We took care to ensure that PC members requesting a lighter workload were assigned fewer papers in each round. Especially in the second round, we manually adjusted reviewers in cases where reviewer confidence was low, based mainly on PC members' bid values.

USENIX ATC '23 had two double-blind rounds of reviews. The goal of the first round was to identify early rejections and also identify for which papers the round 1 reviewers lacked sufficient expertise. In the first round we assigned 3 reviewers per paper, resulting in 1044 reviews. The reviewers had 5 weeks to review papers and 2 weeks for asynchronous online discussion. We notified authors of papers rejected in round 1 (216/353 = 61%) early (sent on March 22nd) to give these authors more time to prepare a future submission.

In the second round, we assigned at least two additional reviewers to the 132 submissions not rejected in round 1, amounting to 284 additional reviews, bringing the total number of reviews for round 2 papers to 5. The reviewers had 3.5 weeks to review these additional papers followed by 3 days of asynchronous online discussion to identify the most important questions for authors to respond to. Authors had 3 days to write a recommended 500 word response with a limit of 1000 words. Almost all authors wrote responses within this range. The reviewers continued the asynchronous online discussion for 1.5 weeks, converging to pre-accept, pre-reject, and pc-discussion decisions on papers. 48 papers were pre-accepted, leaving more discussion time for controversial papers at the synchronous virtual PC meeting. 37 papers were selected for discussion at the PC meeting, of which (22/37 = 59%) of papers were accepted.

We held a two-day synchronous virtual PC meeting with the goals of providing a high-bandwidth channel to resolve discussions, exposing all PC reviewers to a broader set of papers to level-set on quality, raising broader issues that may span multiple submissions and ultimately selecting the final program. We used Zoom, managing conflicts with breakout rooms and HotCRP to manage the discussion order. To manage timezones, we split the paper discussions into 2-hour blocks based on a Doodle poll of timezone availability for the specific reviewers on each paper. PC members were encouraged to attend all sessions regardless of whether one of their papers was being discussed. While it remains a challenge to have full PC participation for the entire meeting due to the wide range of timezones, we found the discussions to be lively and effective.

The PC selected 65 papers for an 18% acceptance rate. 11 were deployed systems papers, 3 were short, and the other 51 were full length research papers. Acceptance was based on the quality of the submissions; in-person conference constraints had no bearing on our decisions. After selecting the program, the program chairs selected two best papers based on nominations from the PC.

Artifact Evaluation Process

USENIX ATC '23 continued to run a joint artifact evaluation process with OSDI, led this year by Jianyu Jiang, Nathan Rutherford, and Cesar A. Stuardo. The artifact evaluation committee chairs assembled a committee consisting of 106 members. The authors of all accepted papers were invited to submit an artifact for an evaluation. 41 out of the 65 USENIX ATC papers (63%) did so. 98% of artifacts received an "Available" badge, 85% received a "Functional" badge, and 63% received a "Reproduced" badge. 61% of papers received all three badges (some artifacts were reproduced, but are not available).

Daniel Porto, researcher at INESC-ID in Lisbon, Portugal, passed away on April 29, 2023. He was a very talented and dedicated researcher, and a recognized expert in the areas of distributed systems and Byzantine consensus. His research led to several impactful publications in venues like EuroSys, OSDI, or DSN, and he served with a sense of community in several committees including the 2023 USENIX ATC/OSDI artifact evaluation committee. As a human being, Daniel will be remembered as an extraordinary person with an endless desire to help others. He touched the lives of many with his incredible generosity and kindness.

Acknowledgements

More than 200 people have contributed to the organization of the USENIX ATC '23, most of them in a voluntary capacity. We would like to thank each and every one of them. We are tremendously grateful to the program committee members for a job extremely well done, and for their personal sacrifices. We thank the Artifact Evaluation committee and the Artifact Evaluation Committee Chairs for their work and contribution, which improves our community and enables future research. Last, we thank the USENIX organization, the USENIX ATC steering committee and OSDI '23 co-chairs. The amount of work and preparation that goes into organizing a conference is immense, and we were astounded by the help and support provided by everyone involved.

Julia Lawall, *Inria*
Dan Williams, *Virginia Tech*
USENIX ATC '23 Program Co-Chairs

2023 USENIX Annual Technical Conference
July 10–12, 2023
Boston, MA, USA

Monday, July 10

Security and Privacy

Searching Graphs

Deduplication

Structuring Graphs

Placement and Fault Tolerance

Updating Code

Tuesday, July 11

Serverless

Troubleshooting and Measurement

Cloud and Microservices

Distributed Storage

Hardware and Software for Security and Performance

Networking

Memory-Related Hardware and Software

Deployed Networking

Key-Value Stores

Security: Attacks

Wednesday, July 12

Virtual Machines

Persistent Memory

Offloading and Scheduling

Kernel and Concurrency

Optimizing ML

GPU

Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines

Dingji Li[1,2,3], Zeyu Mi[1,2]✉, Chenhui Ji[1,2], Yifan Tan[1,2], Binyu Zang[1,2], Haibing Guan[4], and Haibo Chen[1,2]

[1]*Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*
[2]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*
[3]*MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University*
[4]*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University*

Abstract

Existing confidential VMs (CVMs) experience notable network performance overhead compared to traditional VMs. We present the first thorough performance analysis of various network-intensive applications in CVMs and find that the *CVM-IO tax*, which mainly comprises the bounce buffer mechanism and the packet processing in CVMs, has a significant impact on network I/O performance. Specifically, the *CVM-IO tax* squeezes out virtual CPU (vCPU) resources of performance-critical application workloads and may occupy more than 50% of CPU cycles. To minimize the *CVM-IO tax*, this paper proposes Bifrost, a novel para-virtualized I/O design that 1) eliminates the I/O payload bouncing tax by removing redundant encryption and 2) reduces the packet processing tax via pre-receiver packet reassembly, while still ensuring the same level of security guarantees. We have implemented a Bifrost prototype with only minor modifications to the guest Linux kernel and the userspace network I/O backend. Evaluation results on both AMD and Intel servers demonstrate that Bifrost significantly improves the performance of I/O-intensive applications in CVMs, and even outperforms the traditional VM by up to 21.50%.

1 Introduction

As more and more data-processing applications [9, 13, 14, 57] embrace the cloud, widespread concerns are being raised about the security and privacy of data in-use on the cloud. To address these concerns, various confidential computing solutions have been proposed to safeguard data from unauthorized parties. Among them, confidential virtual machine (CVM) solutions, such as AMD SEV [2, 3], Intel TDX [32] and ARM CCA [8], run guest operating systems (OSes) in hardware-isolated environments. In these environments, the complex virtualization stack, such as hypervisor and host OS, is no longer trusted and cannot access data in guest OSes arbitrarily, while still providing resource management functions. This CVM abstraction transparently protects user workloads without requiring any modifications and integrates easily into the existing cloud infrastructure. Therefore, it has gained popularity and is increasingly deployed in data centers.

Unfortunately, while the speed of modern network devices continues to grow (Terabit Ethernet [60] like NVIDIA 400Gbps NIC [51]), the security protections introduced by existing CVM solutions have a significant negative impact on network performance. This paper first conducts a series of experiments to thoroughly analyze the network I/O performance of CVMs. We evaluate widely-deployed network-intensive applications in an AMD SEV-ES/SNP server and a simulated Intel TDX server. The results demonstrate that CVM's security protections significantly increase the *CVM-IO tax*, which we define as the CPU resources used during CVM's I/O procedure, resulting in up to 29% overhead over a traditional VM that does not use any CVM protections. The *CVM-IO tax* is caused by both security protections and intrinsic network I/O procedures in CVMs, draining substantial CPU resources from diverse application workloads.

Concretely, there are three common components in the CVM-IO tax: ① **VM exits** consume up to 11.54% more CPU cycles than the traditional VM. The time consumption of VM exits is greatly increased due to the security checks and protections from the trusted modules (e.g., AMD-SP [3], Intel TDX module [31]) and making the guest aware of emulation events (e.g., AMD #VC [3], Intel #VE [29]). ② The **bounce buffer mechanism**, an I/O staging memory shared between the CVM and hypervisor, takes up to 19.45% CPU cycles for bouncing packets (including headers and payload). I/O operations that could previously be done directly by the hypervisor to the traditional VMs must now be assisted by the bounce buffer mechanism in guest OSes. For example, to emulate a virtual NIC, the hypervisor in traditional VM systems can forward packets between the guest OS and the host network stack by directly copying I/O data to/from the guest private memory. But hypervisors in CVM systems require the guest OS to bounce packets to/from a hypervisor-visible shared memory region due to the memory encryption, introducing I/O data copy overhead. ③ The **packet processing** also spends up to 36.14% CPU cycles preparing payloads from massive network packets for application workloads. The

✉Corresponding author: Zeyu Mi (yzmizeyu@sjtu.edu.cn).

higher the number of packets transferred to the network stack, the more vCPU resources a CVM requires to process their headers. Fortunately, the cost of VM exits becomes negligible when the posted interrupt [59] feature is supported by the hardware, leaving the bounce buffer mechanism and packet processing as the main components of the CVM-IO tax.

This paper aims to **reduce as much CVM-IO tax as possible** for I/O-intensive applications in CVMs by bypassing the bounce buffer mechanism and offloading the packet processing. A straightforward design to bypass the bounce buffer mechanism is to keep the packet content in place by dynamically adjusting the accessibility of the same memory region to the hypervisor. However, this approach is limited by the memory encryption hardware support [34], which does not allow the plaintext contents of a memory region to be preserved when modifying the accessibility of the memory region [29]. To reduce network packet processing cost, the existing design is to pass fewer packets to the network stack by reassembling multiple small packets into a large one in the guest device driver. But the guest device driver still has to process a large number of packets, consuming substantial CPU resources.

Fortunately, there are three observations that can help us address the challenges mentioned above. We observe that either end-to-end encryption or a CVM's private memory alone can protect data security, while applying both protections to the payload is redundant. Additionally, we notice that end-to-end encryption/decryption can also change the payload's memory location, which is functionally equivalent to bouncing between two memory regions. As a result, bypassing payload bouncing can be achieved by directly encrypting/decrypting the payload into/from the guest-host shared memory. Another observation is that the network I/O backend typically has plenty of residual CPU resources. Given the bottleneck experienced by the saturated vCPUs of network-intensive CVMs, an opportunity arises to offload packet processing to the network I/O backend. This approach effectively utilizes the available CPU resources, alleviating the strain on vCPUs and resulting in improved performance.

Based on these observations, this paper proposes **Bifrost**[1] to improve the paravirtual network performance of the CVM with three techniques: ① The *zero-copy encryption deduplication* eliminates payload bouncing by leveraging dedicated guest-host shared memory to remove redundant encryptions on the payload in a zero-copy way. When receiving packets, the end-to-end encrypted payload is directly decrypted from the shared memory. When sending packets, the payload is directly encrypted into the shared memory. To minimize modifications, the shared memory is in the form of dedicated non-uniform memory access (NUMA) [37] nodes, allowing memory allocators in the guest kernel to be reused. ② The *one-time trusted read* mechanism protects guest OSes from time of check to time of use (TOCTTOU) attacks while accessing packets in the dedicated shared memory. With these two techniques, the bouncing of the end-to-end encrypted payload, which takes up much CPU resources of CVMs, is securely bypassed. ③ The *pre-receiver packet reassembly* reduces vCPU resources utilized by the device driver by offloading the task of reassembling received packets to the network I/O backend. Thus, CVMs are able to process fewer packets with larger payload, reducing the packet processing cost on vCPUs.

We have implemented a Bifrost prototype by modifying the guest OS kernel and host user-level software. The prototype extends the Linux v6.0-rc1 kernel in the guest OS with 815 lines of code, and adds 175 lines to OpenvSwitch v2.17.3 and 541 lines to DPDK v21.1.2, both of which run in the host user mode. We have also evaluated Bifrost's performance on both AMD and Intel platforms. The results show that, with advanced posted interrupt support, Bifrost enhances the performance of I/O-intensive applications in CVMs, surpassing traditional VMs by up to 21.50%.

In summary, this paper makes the following contributions:

- The first thorough performance analysis of I/O-intensive applications in CVMs on existing and next-generation hardware platforms, revealing their bottlenecks and overhead sources compared to traditional VMs.
- A secure paravirtual I/O design that greatly reduces the CVM-IO tax, significantly improving the performance of I/O-intensive applications in CVMs.
- A Bifrost prototype and a comprehensive evaluation on AMD and Intel platforms, demonstrating improvements on existing and future CVM hardware. The prototype is available at `https://github.com/IPADS-Bifrost`.

2 Background

2.1 Confidential VMs (CVMs)

There are different CVM solutions based on specialized hardware extensions. All of these solutions leverage hardware memory encryption and integrity checking [30, 34] to enforce confidentiality and integrity. They share the same CVM abstraction that excludes the entire virtualization stack from the trusted computing base (TCB). As shown in Figure 1, the trusted firmware, which is the unique software TCB, isolates the CVM from untrusted hypervisors and traditional VMs.

Existing CVM systems typically divide the physical memory of a VM into two major security types: private memory and shared memory. The private memory is encrypted by hardware and cannot be accessed or modified by any untrusted entities outside the VM, while the shared memory holding plaintext data can be accessed by the hypervisor. The CVM systems also allow the hypervisor to switch private memory and shared memory to each other at runtime. However, the data content of the memory page cannot be preserved before and after the security type switch [29]. Hence, the guest must

[1] Bifrost, the rainbow bridge from Norse mythology, metaphorically represents the secure and rapid transfer of CVM's I/O data (Asgard's gods) to and from the untrusted hypervisor (Midgard).

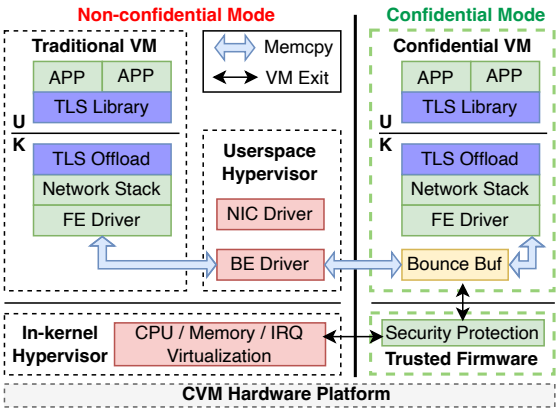


Figure 1: The paravirtual I/O networking architecture of traditional VMs and CVMs atop the CVM hardware platform. The black arrows represent the path of VM exits and VM enters. The light blue arrows indicate memory copies that consume CPU resources. The FE Driver and the BE Driver in the figure represent drivers in frontend and backend, respectively.

move data outside of private memory before the security type switch, and then copy it back to the new shared memory. Besides, it takes much effort to finish the security type switch. The guest OS has to cooperate with the host hypervisor to alter address translation data structures and maintain CPU micro-architectures, requiring multiple VM exits and interprocessor communications [4, 29]. As a result, the security type switch is unsuitable to occur frequently in CVMs.

2.2 Paravirtual I/O Networking in CVM

Paravirtual I/O has become a primary I/O virtualization choice for modern cloud providers owing to its high performance and excellent compatibility. There are two cooperative drivers in paravirtual I/O, a frontend driver in the guest VM and a backend driver in the hypervisor, which communicate with each other through shared memory. To provide maximum network performance, the backend driver can be deployed in the host userspace, for instance, using vhost-user [46, 53], to directly control the device in a busy-polling mode [16].

An example of paravirtual I/O networking of the traditional VM is shown in the left part of Figure 1. The applications in the userspace deal with payload, while the network stack and the frontend driver in the kernel handle packet processing. The packet processing includes network functions that handle conversions between payload and packets. For example, in the transmission (TX) direction, the payload from applications and the headers from the network stack are encapsulated into network packets, after which the backend driver is notified to send them out. Because the hypervisor can access the entire memory space of a traditional VM, the backend driver can copy the packets freely from the guest memory to its own memory and forwards them to the NIC driver.

Memory pages in CVMs, including those containing packets, are set to private by default. However, the host OS is untrusted and cannot access the private memory of CVMs (see § 2.1). To allow the host OS to transfer packets, CVMs utilize a bounce buffer mechanism that sets up a guest-host shared memory as an intermediary. As shown in the right part of Figure 1, the guest OS reserves a shared memory region with the host OS as the bounce buffer and copies the outgoing packets to it. Afterwards, the backend driver can copy the packets to the hypervisor as normal. As a result, the bounce buffer leads to excessive memory copies for I/O virtualization.

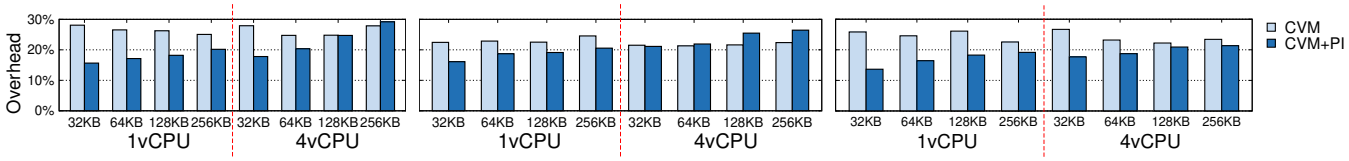
2.3 Transport Layer Security (TLS)

TLS is an end-to-end security protocol designed to protect data in transit by leveraging cryptography. It has been commonly used by modern applications to secure their I/O payload in transit [6, 17, 49, 55, 61]. CVM solutions have made it a mandatory requirement for their applications [22, 26, 54]. Moreover, today's OSes, such as Linux, provide in-kernel TLS support, enabling userspace applications to offload TLS to the kernel for enhanced performance and expanded features [19]. As shown in Figure 1, in-kernel TLS allows the payload from the page cache to be encrypted without going through the userspace.

The industry currently implements the TLS protocol based on encryption algorithms such as AES-GCM [18] to assure the confidentiality and integrity of data simultaneously. The output of these encryption algorithms consists of encrypted ciphertext for confidentiality, and an authentication tag generated from the ciphertext for integrity. To provide complete data security protection, the correctness of both the ciphertext and its authentication tag must be guaranteed during encryption, and vice versa.

2.4 Exitless Interrupt Virtualization

In the paravirtual I/O networking scenario, when a virtual NIC (i.e., network backend) receives some network packets, it notifies the guest VM with a virtual interrupt. The guest VM then needs to interact with the virtual interrupt controller to perform Acknowledgment (ACK) and End of Interrupt (EOI). Traditional techniques rely on the hypervisor to emulate interrupt delivery and interrupt controller access of guest VMs using trap-and-emulate approaches. However, virtualizing interrupts in this way can be a significant source of overhead, as each virtual interrupt's completion necessitates multiple VM exits and entries. To address this issue, modern hardware platforms have introduced the posted interrupt technique to enable exitless virtual interrupt delivery. They have also extended their interrupt controllers with specialized virtualization support to eliminate VM exits caused by ACK and EOI. Interrupt controllers with virtualization extensions are currently in production by all mainstream hardware vendors such as Intel, AMD, and ARM. Full-featured posted interrupt is available on the Intel platform, and it will soon be supported on other platforms (e.g., next-generation products with AMD AVIC [4] and ARM GICv4 [7]).



(a) Memcached **(b)** Nginx **(c)** Redis

Figure 2: Normalized overhead compared with traditional VM in I/O-intensive applications. The Y-axis is the normalized overhead compared to the baseline of each group. CVM+PI represents CVM + Posted Interrupt.

3 Analysis of CVM-IO Tax

In this section, we quantify the performance impact of the CVM-IO tax by comparing the I/O performance of existing CVMs with that of traditional VMs. The CPU execution time of I/O-intensive applications running in a CVM can be divided into two parts: 1) *Application workloads*: the time spent on executing application workloads, including business logic and payload processing. 2) The *CVM-IO tax*: the time spent on CVM-specific security protections and intrinsic network I/O procedures. It consists of VM exits, the bounce buffer mechanism, and the packet processing during the payload preparation for application workloads.

All experiments are conducted on a 128-core AMD SEV-ES/SNP server and a 24-core Intel server with 200Gbps NICs. The AMD server is used to evaluate the I/O performance of real CVMs, referred to as ***CVM***. However, the AMD server does not support posted interrupt, resulting in degraded performance due to numerous VM exits during virtual interrupt deliveries. Therefore, we simulate next-generation CVMs using the Intel server that supports posted interrupt, named ***CVM+PI***. More detailed testbed and simulation configurations are described in § 7.1. For fair performance comparison, *CVM*'s baseline is the vanilla AMD traditional VM, while *CVM+PI*'s baseline is the vanilla Intel traditional VM.

To achieve optimal network performance, we choose vhost-user as the network backend in follow-up experiments. We still use the SEV-ES VM because the SEV-SNP VM does not support vhost-user due to its lack of huge page support. Theoretically, the security protections introduced by SEV-SNP do not further increase the CVM-IO tax.

3.1 CVM-IO Tax Breakdown

We first evaluate the overall performance using three representative network-intensive applications: Memcached and Redis for key/value stores, and Nginx for web servers. All applications and benchmarks enable the in-kernel TLS support for end-to-end protection. Figure 2 depicts the normalized performance overhead of CVMs compared to their respective baselines. In all three benchmarks, *CVM* incurs 21%-28% overhead, while *CVM+PI* exhibits 13%-29% performance degradation. As a result, the performance impact of CVM-IO tax results in significant overhead over baselines.

We further take the Memcached benchmark with the 4vCPU-256KB test cases as an example to break down the time cost of the CVM-IO tax of CVMs, as shown in Figure 3. In the left subfigure, the *VM Exit*, *Bouncing* and *Pkt Processing* denote the three corresponding parts of the *CVM-IO tax* mentioned earlier. Because all vCPUs are fully utilized during the benchmark, the impact on overall performance becomes more significant as the percentage of CPU time consumed by the CVM-IO tax grows. The CVM-IO tax consumes over 50% of the total CPU time in all CVMs during the benchmark. For *CVM*, VM exits taking up more than 20% CPU time have more impact than the bounce buffer. For *CVM+PI*, the bounce buffer cost occupies more than 19% CPU time, while VM exits take up a tiny percentage of the total time in both *CVM+PI* (less than 1.2%) and its baseline (less than 0.5%). Packet processing in all the above cases accounts for about 30% of CPU time and thus has a considerable impact.

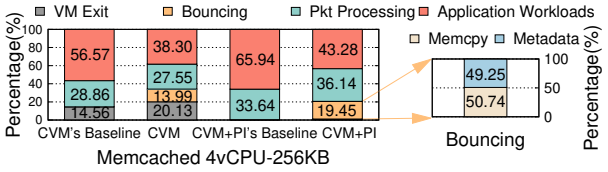


Figure 3: CPU time breakdown of the Memcached 4vCPU-256KB case in CVMs. The left subfigure shows the CPU time breakdown of *CVM*, *CVM+PI* and their baselines in the Memcached 4vCPU-256KB case. The right subfigure shows the CPU time breakdown of the bounce buffer part in *CVM+PI*.

The overhead over baselines can be attributed to the reduction in CPU time of application workloads due to the CVM-IO tax. For example, in the 4vCPU-256KB case of *CVM+PI*, the CVM-IO tax leaves 34.35% fewer cycles for application workloads than the baseline, explaining the 27.44% overhead. Since packet processing in both CVMs and their baselines consumes a similar portion (about 30%) of CPU time, VM exits and the bounce buffer in CVMs contribute the most to the overhead.

Take-away I

> *The CVM-IO tax that occupies more than half of total CPU time incurs a substantial performance impact on CVMs. VM exits and the bounce buffer are the primary sources of overhead over baselines.*

Lengthy VM Exits The AMD SEV-ES hardware introduces protection for CPU states (e.g., registers) of each CVM against the untrusted hypervisor during VM exits. In contrast

to traditional VMs, this protection adds thousands of cycles to each VM exit. We first break down the VM exit handling cost during every virtual interrupt delivery, finding that, on average, *CVM* spends 7,476 cycles on guest-host world switches, whereas a traditional VM only spends 1,643 cycles. We then collect the number of VM exits per second during the Memcached 4vCPU-256KB benchmark for different CVMs. *CVM* averagely triggers 41,615 VM exits per second on each vCPU, while *CVM+PI* only triggers 2,803 VM exits per second on each vCPU, an order of magnitude less than *CVM*.

The results indicate that frequent VM exits taking up more than 20% of total CPU time have a significant impact on *CVM*. However, with posted interrupt support (*CVM+PI*), the impact of VM exits is almost negligible. Fortunately, all next-generation CVM platforms, including AMD SEV, Intel TDX and ARM CCA, support posted interrupt, so that the performance impact of VM exits can become minimal.

Take-away II

> *VM exits may take up a large portion of the CPU time of CVM due to their high frequency and latency, but their performance impacts can become minimal with the posted interrupt support on next-generation hardware.*

Bounce Buffer To analyze the overhead of bounce buffers, we break down the CPU time spent on bounce buffers in the 4vCPU-256KB case of *CVM+PI* into two parts: copying I/O data (packets in this case) and maintaining metadata for buffer allocation and freeing. The breakdown result shown in the right subfigure of Figure 3 indicates that I/O data copy (corresponding to the *Memcpy*) consumes 50.74% of the bounce buffer time, while the metadata maintenance (corresponding to the *Metadata*) spends 49.25% of the bounce buffer time. Besides, the experimental results of small and large data sizes reflect that the performance impact of the bounce buffer rises as the data size increases.

Take-away III

> *The bounce buffer consumes a large percentage of CPU resources due to I/O data copying and metadata maintenance. It is necessary to avoid bouncing large-size I/O data to minimize the bounce buffer's performance impact.*

Packet Processing Packet processing in both the frontend driver and the network stack consumes up to 36.14% of CPU time in Memcached 4vCPU-256KB cases. Since the packet processing time cost is proportional to the number of packets processed, the large number of packets in I/O-intensive scenarios can demand a significant amount of CPU resources.

Take-away IV

> *Packet processing occupies a large fraction of CPU time due to the large number of packets to be processed. Reducing the number of packets to be processed can mitigate its performance impact.*

3.2 Summary

To sum up, our experiments have demonstrated that CVMs incur up to 29% overhead in I/O-intensive applications compared with traditional VMs. On the next-generation hardware with posted interrupt support, the tax of VM exits becomes negligible while the bounce buffer tax has a more significant impact on CVMs. Additionally, the packet processing tax consumes a great portion of CVMs' vCPU resources due to the large number of packets, which is also the case for traditional VMs. Therefore, it is essential to reduce the cost of the bounce buffer as well as the packet processing in CVMs to minimize the CVM-IO tax and achieve high network performance.

4 Overview

4.1 Design Goals

The primary goal of Bifrost is to reduce the paravirtual I/O network tax of existing CVM solutions while maintaining the same level of security guarantees. Besides, the design of Bifrost should be general enough to be easily applied to CVM solutions on various platforms, such as x86, ARM and RISC-V, and to support different host and guest OSes, including Linux, FreeBSD and Windows. Further, it is demanding that Bifrost should avoid intrusive modifications to existing software stacks and keep transparent to userspace applications in CVMs to make it practical for real-world scenarios.

4.2 Challenges

To reduce the CVM-IO tax, Bifrost should optimize the bounce buffer mechanism and the packet processing procedure. However, it is not easy to implement these optimizations due to the following two technical challenges:

C1: Out-of-place hardware encryption and decryption. The ideal way to eliminate the bounce buffer mechanism for a network packet is to enable zero copy by maintaining the packet within the same memory region throughout its entire lifecycle. In addition, either the guest or the host should have exclusive access to the packet's memory region while processing it to ensure data security, necessitating memory security type switches at runtime. However, as mentioned in § 2.1, when a private page containing a packet is converted to a shared page (and vice versa), the packet in this page is lost and unable to be correctly passed to the hypervisor. Moreover, changing the security type of guest memory pages is too expensive to be a frequent operation on the I/O critical path.

C2: Costly packet pre-processing in the device driver. Packet processing primarily operates on packet headers rather than payloads. To minimize the cost of packet processing, a commonly employed technique is to pre-process multiple small packets within the same flow into larger packets before submitting them to the network stack. Nonetheless, the virtual NIC driver still has to occupy large quantities of vCPU resources to handle massive small packets coming from the high-speed NIC.

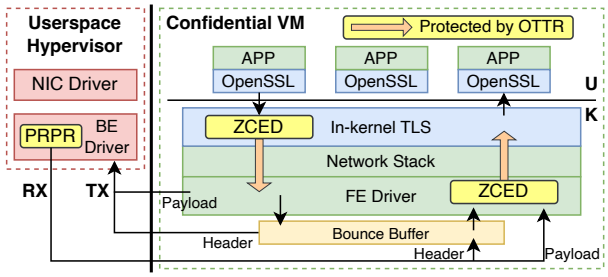


Figure 4: The overall architecture of Bifrost.

4.3 Observations and Insights

We observe three characteristics of existing CVM systems, allowing us to propose new designs that are appropriate for CVM scenarios to address the above challenges.

O1: Either private memory or end-to-end encryption alone is sufficient to assure data security. Data security can be ensured by either private memory protection or end-to-end encryption. Besides, it is not always better to apply multiple security protections to a piece of data at the same time, especially for performance-critical I/O data. Specifically, the guest OS in current CVM solutions initially encrypts the payload into private memory. But private memory protection is a redundant security mechanism for data that has already been encrypted. As a result, the payload bouncing tax can be eliminated by retaining existing end-to-end encryption while removing private memory protection at the same time.

O2: End-to-end encryption has the side effect of moving the memory location of the payload. The procedure of adding end-to-end encryption allows CVM to relocate payloads to a different memory location. Hence, end-to-end encryption at the in-kernel TLS layer provides an opportune moment to also remove private memory protection on the payload for userspace applications. In particular, the generated ciphertext during encryption can be directly written to the target shared memory with the host, ensuring data security while eliminating unnecessary copies and bouncing overhead.

O3: I/O backends usually have plenty of residual CPU resources available. Modern virtualization systems usually leverage dedicated CPUs to run I/O backends of VMs for high and predictable I/O performance [16, 45]. Unlike CPUs running CVMs' vCPUs, which are likely to be fully loaded due to complex in-guest logic, those running I/O backends have less work to do and thus have plenty of free CPU resources. As a result, I/O backends with adequate residual CPU resources can be utilized to release the burden of vCPUs by pre-processing packets before passing them to CVMs.

4.4 Architecture and High-Level Design

Based on the above observations, Bifrost leverages the side effect of end-to-end encryption and the residual CPU resources of network I/O backends to eliminate payload bouncing and reduce packet processing cost in CVMs in an application-transparent way. Figure 4 shows the architecture of Bifrost.

To address challenge **C1**, Bifrost proposes two designs to enable zero-copy transparently and securely for the end-to-end encrypted payload in the CVM. **D1:** *zero-copy encryption deduplication* (§ 5.1) eliminates payload bouncing by keeping the end-to-end protected payload in the same shared memory during its lifetime. Specifically, Bifrost directly stores output from the in-kernel TLS layer to the guest-host shared memory without copying any payload, and vice versa. To minimize modifications, Bifrost creates dedicated NUMA nodes to serve as shared memory for this design, so that memory allocators in the guest kernel can be reused. However, concurrent memory accesses to plaintext data in shared memory may lead to TOCTTOU attacks. A malicious host can tamper with data that has passed the security checks of the guest OS, such as altering a packet header after it has passed the checksum check. To defend against this attack, Bifrost introduces **D2:** *one-time trusted read* (§ 5.2), which ensures that the guest OS can only read and trust the target data content from shared memory once, as additional reads from the same memory may lead to host-tainted data content. The guest must process data after it has been read into registers or private memory to defend against host tampering during guest processing, thus eliminating TOCTTOU issues.

To address challenge **C2**, Bifrost proposes another design to complete pre-processing network packets before they reach the CVM. **D3:** *pre-receiver packet reassembly* (§ 5.3) makes use of the network backend's free CPU resources to pre-process multiple small incoming packets into a large one before transmitting them to the guest OS.

As mentioned in § 3.2, while **D1** and **D2** are designed to optimize payload bouncing issues that are specific to CVMs, **D3** can also be leveraged to reduce packet processing cost in traditional VMs.

We explain the Bifrost architecture and its design points by describing the high-level workflows of packet receiving and sending. **Packet receiving workflow:** When a network packet carrying an end-to-end encrypted payload arrives at the network I/O backend, Bifrost attempts to merge it with other same-flow packets, if possible, by pre-processing the packet with PRPR (**D3**). Then Bifrost flushes those pre-processed network packets to the frontend driver through virtual network queues of the CVM. The zero-copy aware TOCTTOU defense (**D2**) in the frontend driver only copies small metadata such as packet headers to private memory for security, while keeping the end-to-end encrypted payload in the shared memory allocated from dedicated NUMA nodes (**D1**). Next, the frontend driver constructs basic data structures (e.g., *skbuff* in Linux) for these pre-processed incoming packets before passing them to the network stack. Afterwards, Bifrost utilizes the in-kernel TLS support to decrypt the end-to-end encrypted payload directly from shared memory into the application's private memory. As a result, the packet receiving workflow experiences no end-to-end encrypted payload bouncing and less packet processing cost in the CVM.

Packet sending workflow: When an application begins to

send out a payload from the guest OS, Bifrost first leverages the in-kernel TLS support to encrypt plaintext from either application memory or kernel page cache in private memory and places the encrypted result directly into the guest-host shared memory allocated from dedicated NUMA nodes (**D1**). The memory copy of the end-to-end encrypted payload from private memory to the shared bounce buffers is removed at this step. For small metadata that is not protected by end-to-end encryption, the zero-copy aware TOCTTOU defense (**D2**) enforces Bifrost to fall back to the bounce buffer mechanism. Consequently, there is no end-to-end encrypted payload bouncing in its sending workflow.

4.5 Threat Model and Assumptions

The threat model of Bifrost is the same as that of existing CVM solutions. The TCB only comprises the CPU hardware and minimized trusted monitor firmware or software, if any. Attackers can control any untrusted software entities or hardware devices to launch attacks on CVMs. Therefore, for a specific CVM, all software outside it, including the hypervisor and other CVMs, and hardware devices, are untrusted. We assume that a CVM does not voluntarily reveal its sensitive data and protects its I/O data with end-to-end encryption. Denial-of-Service (DoS) attacks [11] are out of scope. Although CVM implementations may have bugs [5, 41] and are subject to side-channel attacks [12, 39, 40, 47], we do not consider them because they are orthogonal to this paper.

5 Design and Implementation Details

5.1 Zero-Copy Encryption Deduplication (ZCED)

Bifrost reserves a contiguous shared memory region for paravirtual I/O networking in the guest physical address (GPA) space. This shared memory appears as NUMA nodes dedicated for ZCED (hereinafter called ZCED NUMA), allowing Bifrost to utilize mature memory management mechanisms in existing guest OSes. Moreover, the location and size of ZCED NUMA memory are fixed at the boot time of a CVM for optimal performance.

Boot-time initialization: The memory range of a ZCED NUMA node can be configured by setting the base GPA and total length via the kernel command line. As shown in § 7.4, ZCED NUMA nodes of 200MB can satisfy the demands of all network-intensive benchmarks in our experiments. Bifrost parses the number of ZCED NUMA nodes and adds the specified guest memory range to each node. All ZCED NUMA nodes are created with no associated vCPU. Before a ZCED NUMA node is available for memory allocations, Bifrost sets its memory security type to shared. To achieve optimal performance, proper distances should be specified between NUMA nodes to assist the guest kernel in allocating memory [36]. The distances between ZCED NUMA nodes are the same as those between normal NUMA nodes to which their memory ranges originally belonged, while each ZCED NUMA node is zero distance from its original NUMA node.

Runtime allocation: To prevent data leakage caused by unintentional data store into the ZCED NUMA memory, Bifrost adjusts the memory allocation policies of the guest OS to only allow explicit allocation to acquire ZCED NUMA memory. Hence, the original memory allocations in the system do not allocate from ZCED NUMA nodes, avoiding the security issue of inadvertently exposing sensitive data. Guest kernel components are merely able to allocate memory from ZCED NUMA nodes by assigning a special allocation flag (e.g., a *GFP* flag in Linux) provided by Bifrost to parameters of memory allocation invocations. The allocator will first try to acquire memory from the closest ZCED NUMA node to the vCPU running this component. In the frontend driver, Bifrost checks the memory location in which the payload resides, and if it belongs to a ZCED NUMA node, Bifrost will bypass the bounce buffer mechanism.

In the TX direction, Bifrost modifies the in-kernel TLS layer to transparently intercept communications between upper applications and the lower network stack. The payload in the TX direction must go through *sendmsg* and *sendpage* functions of the existing in-kernel TLS layer before entering the network stack. *sendmsg* is the most often used function for sending payload from userspace, whereas *sendpage* is specialized for transferring payload from the storage (e.g., page cache). Bifrost just adds the special allocation flag to the parameters of memory allocation invocations in these two functions to allocate memory from ZCED NUMA nodes for storing encrypted payload.

In the RX direction, the guest memory regions used to accept incoming packets are allocated and assigned by the frontend driver (i.e., virtio-net in our case). Bifrost modifies the memory allocation invocations for these regions by adding the allocation flag as well. When an application attempts to receive payload, Bifrost decrypts the ciphertext directly from the ZCED NUMA memory to private memory.

5.2 One-Time Trusted Read (OTTR)

To defend against TOCTTOU attacks, Bifrost only trusts the data obtained from the first read of the ZCED NUMA memory during the guest OS's handling of packet headers and end-to-end encrypted payload.

Packet header handling: The content of each packet header should only be used after it has been validated by the guest OS's packet processing functions. However, if a malicious host modifies the header after the guest OS's validation, the guest OS may encounter problems due to the invalid header. For instance, buffer-overflow problems can happen if the guest OS uses a modified length to extract payload from packets.

To prevent this, Bifrost must read a packet header from the ZCED NUMA memory into a private memory region before further processing it. This read only happens once for each packet header, and Bifrost will never read the header from

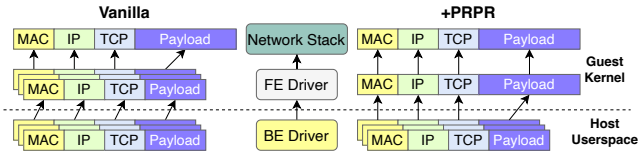


Figure 5: Comparison of the packet reassembly workflow of the vanilla CVM and the CVM with PRPR.

the ZCED NUMA memory again to prevent the host from subsequently tampering with the content of packet headers. In the TX direction, unlike the end-to-end encrypted payload that is stored in the ZCED NUMA memory, network packet headers are placed in private memory. The frontend driver still leverages the bounce buffer mechanism to copy packet headers to the guest-host shared memory before transmitting them to the backend driver. It has a very small impact on performance due to the small size of packet headers.

End-to-end encrypted payload handling: For end-to-end encrypted payload, as mentioned in § 2.3, data security protection requires that the encryption must generate both correct ciphertext (i.e., encrypted payload) and authenticated tag, and the decryption must correctly verify the ciphertext integrity with the authenticated tag. Both the authenticated tag generation and the integrity verification take the ciphertext as input, which exists in ZCED NUMA memory and may be tampered with by the host, resulting in compromised payload integrity. For instance, in the context of zero-copy I/O, the current Linux AES-GCM implementation on the x86-64 platform double reads the ciphertext from the same ZCED NUMA memory in the last phase of parallel decryption, suffering from TOCTTOU attacks on the ciphertext.

To prevent payload TOCTTOU attacks, in the decryption procedure, Bifrost reads only once from the ZCED NUMA memory to load the ciphertext value into CPU registers and always uses the correct ciphertext in the registers afterwards, avoiding reading a potentially compromised ciphertext. Similarly, during the encryption procedure, the ciphertext for each payload is guaranteed to remain valid from the moment it is generated in the register until it exits the register. Thus, Bifrost calculates the authentication tag using the correct ciphertext that is still in the CPU registers.

5.3 Pre-receiver Packet Reassembly (PRPR)

Large packets are split into smaller ones before sending out due to transmission size limit. To save CPU resources consumed by packet handling, prior work [33, 52] has decreased the number of packets passed to the network stack by reassembling small packets into large ones in advance. Modern OSes support small packets coalescing at the device driver layer using GRO [15]. However, packet reassembly in the guest device driver can still consume significant vCPU resources, severely affecting the application performance when handling large numbers of packets. While modern NICs enable hardware coalescing without engaging CPU using LRO [23], it is hard for hardware to dynamically adjust reassembly rules and support new packet formats. Inappropriate coalescing even causes metadata loss and network connection disruptions [15].

In comparison to prior approaches, Bifrost offloads the packet reassembly to the hypervisor backend driver which has sufficient CPU time, freeing up precious vCPU resources for CVMs without sacrificing flexibility. The packet reassembly logic in Bifrost is similar to that of previous work [15] since network packets share the same format.

Overall procedure: When a network packet arrives at the network backend, some packets may be cached in the backend and waiting for reassembly. Bifrost first parses the current packet header to determine if any cached packets from the same flow exist. If present, Bifrost tries to merge the current packet with the cached same-flow ones. Eventually, based on the status information in the currently cached packets in the network I/O backend, Bifrost decides whether it is time to flush them to the frontend driver in the guest OS.

Same-flow packet detection: Same-flow packets are network packets that share the same source, destination and sequence number. As our current implementation focuses on TCP/IP packets, Bifrost first recognizes headers that have the same MAC address, IP address and TCP port in both source and destination directions as same-flow candidates. Then Bifrost regards these candidates with an identical TCP acknowledgment (ACK) number as same-flow packets.

Flexible per-VM flush rules: It is essential to flush packets to the guest OS at an appropriate time since the network performance is highly sensitive to packet latency. When a newly received packet has a cached same-flow packet, Bifrost first checks whether these two packets have consistent status information, such as the time to live (TTL) field. If not, Bifrost flushes the old cached packet to the frontend driver. Otherwise, Bifrost reassembles these two packets into a new one. Finally, Bifrost flushes the new packet if it contains an immediate-flush flag (e.g., the TCP PSH flag). For a received packet that has no same-flow packet, Bifrost directly determines whether to flush it by checking its immediate-flush flag.

In addition to the above basic rules, Bifrost also allows each guest OS to customize flush rules. Bifrost provides paravirtual interfaces for receiver CVMs to install their own rules to disable reassembly, adjust the maximum number and timeout of cached packets.

Packet reassembly: Among the cached same-flow packets, the currently received packet can only be reassembled with the packet whose payload is contiguous with it. Bifrost finds neighbors of the received packet for reassembly by comparing their TCP sequence (SEQ) numbers. As depicted in Figure 5, duplicate packet headers are merged during reassembly.

6 Implementation Complexity

We implement a Bifrost prototype using Linux as the guest kernel and OpenvSwitch-DPDK as the network I/O backend.

In the Linux v6.0-rc1, we introduce 815 lines of code to support ZCED and OTTR. These changes include initializing

ZCED NUMA nodes during memory subsystem bootstrapping, replacing memory allocation invocations, and defending against TOCTTOU risks in AES-GCM assembly.

In the DPDK v21.11.2, we add 541 lines of code to implement PRPR, which primarily consists of two parts: 1) Reorganization of network packets, including header trimming during packet reassembly, flag resetting, etc. 2) Flush rules, which mainly focus on deciding whether to cache or immediately flush an incoming packet. Our implementation also adds 175 lines of code to OpenvSwitch v2.17.3, which preprocesses the network packets by parsing headers in advance, and invokes the interfaces provided by the DPDK.

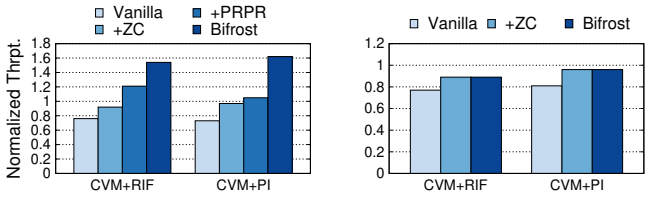
7 Evaluation

7.1 Experimental Setup

Testbed: Our testbed remains the same as in § 3, consisting of an AMD server and an Intel server running Ubuntu 20.04.4 LTS. The AMD server has two 64-core AMD EPYC 7T83 CPUs at 2.45GHz (128 cores in total) and 500GB DDR4 DRAM. The Intel server has two 12-core Intel Xeon Gold 5317 CPU at 3.00GHz (24 cores in total) and 188GB DDR4 DRAM. Both machines are equipped with one single-port Mellanox Connect-X6 200Gbps NIC and are back-to-back connected with a fabric cable. We disable CPU frequency boost features to lessen performance data fluctuation. The AMD server's host kernel is Linux v5.19.0-rc6 with SEV-ES and SEV-SNP support, while the Intel server's host kernel is Linux v5.4.0. The guest kernel version of all CVMs and their baseline VMs is Linux v6.0-rc1. Each guest OS is assigned with either 1 vCPU or 4 vCPUs, 16GB memory and a 2-virtqueue virtio-net device backed by the vhost-user backend based on OpenvSwitch v2.17.3 and DPDK v21.11.2. For each benchmark, the server side runs in the guest OS while the client side runs in the host OS on the other server.

To avoid the interference of unintended scheduling or interrupts, we isolate 6 cores on each server. The CPU isolation is achieved by the *isolcpus* function in the Linux kernel, and the binding is done by the *qemu-affinity* command for vCPUs and *pmd-cpu-mask* parameter for the OpenvSwitch-DPDK-based vhost-user backend. Each thread of vCPUs and the vhost-user backend is pinned to a different isolated CPU. IOMMUs of both machines are set to passthrough mode.

Naming Convention and Configurations: As mentioned in § 3, ***CVM*** represents real SEV-ES/SNP CVMs on the AMD server, while ***CVM+PI*** represents simulated TDX CVMs with posted interrupt enabled on the Intel server. The simulation is based on a vanilla Intel traditional VM, which further enables bounce buffer (i.e., Linux SWIOTLB [62]) for virtio devices and adds an additional 10,000 cycles to each VM exit to simulate the cost of guest-host world switches. The simulated world switches consume 2,524 more cycles than that of the SEV-ES/SNP VM. To the best of our knowledge, SEV-ES/SNP's lengthy VM exits primarily result from uncore co-processor (i.e., AMD-SP) intervention, whereas TDX's VM exits solely involve in-core firmware, leading to lower latency. As a result, though the increased VM exit latency does not guarantee identical performance to real Intel TDX hardware, the simulated VM exit overhead should be no less than that of real TDX VMs.



(a) RX **(b)** TX

Figure 6: TLS normalized throughput on both AMD and Intel server. The Y-axis is the normalized throughput compared with the baseline. *+PRPR* is not shown in (b) because it does not provide benefits for TX performance.

To get the performance of *CVM* close to the SEV-ES VM with posted interrupt, we optimize AMD *CVM* with reduced VM exit frequency by modifying the network backend to lower its notification frequency to the guest OS. We call the optimized version ***CVM+RIF***, signifying *CVM + Reduced Interrupt Frequency*. *CVM+RIF*'s baseline is the vanilla AMD traditional VM with reduced VM exit frequency, while *CVM+PI*'s baseline is the vanilla Intel traditional VM. To show the individual contribution of Bifrost's each technique, ***+ZC*** denotes only applying the ZCED as well as the OTTR, while ***+PRPR*** indicates adding the PRPR alone. The PRPR is configured to cache up to 1024 TCP flows for 2 virtqueues. Each flow can hold up to 1024 packets, and at most 32 packets can be submitted to the I/O frontend simultaneously.

7.2 Performance Improvement

In this section, we focus on the performance improvement of *CVM+RIF* and *CVM+PI*. We first build a microbenchmark to investigate the upper bound on the performance improvement that Bifrost can bring to I/O-intensive applications and then study the performance gains of real-world applications from Bifrost's design. Since the posted interrupt hardware has been able to minimize the performance impact of VM exits, we concentrate on Bifrost's effect on CVMs atop such hardware.

7.2.1 Microbenchmark

We develop a TCP-based TLS client/server pair to evaluate the network throughput. They simply contain simple code for single-threaded I/O data sending and receiving. This minimizes the time cost of business logic, demonstrating the maximum possible application performance improvement. To fully saturate the vCPU like an I/O-intensive application, we run 4 TLS server instances in a 1-vCPU VM.

RX Throughput: Figure 6a shows the network throughput comparisons in the RX direction. *CVM+RIF* attains 5.26 Gb/s, which is 24.10% slower than its baseline's 6.93 Gb/s. With the ZCED, *+ZC* alone (6.38 Gb/s) can reduce the slowdown to

7.81%. With the PRPR, *+PRPR* itself (8.39 Gb/s) can outperform the baseline by 21.10%. By combining both techniques, Bifrost reaches 10.64 Gb/s, which is 53.55% higher than the baseline. For *CVM+PI* (7.48 Gb/s), it incurs 27.03% overhead than its baseline (10.26 Gb/s). The throughput is increased to 9.99 Gb/s in *+ZC* (2.59% overhead) and grows to 10.76 Gb/s in *+PRPR* (4.95% better). Integrating both techniques makes Bifrost reach 16.64 Gb/s, which is 62.20% higher than the baseline. Therefore, Bifrost can boost performance by up to 89.23% (from 27.03% overhead to 62.20% better than the baseline) for applications experiencing high RX traffic in existing CVMs.

TX Throughput: Figure 6b shows the throughput comparisons in the TX direction. *CVM+RIF* attains 9.59 Gb/s, which is 23.03% slower than its baseline's 12.45 Gb/s. *+ZC* (11.04 Gb/s) reduces the slowdown to 11.37%. Bifrost has 10.79% overhead (11.11 Gb/s), slightly better than *+ZC*. Experiments of *CVM+PI* yield similar results. Therefore, Bifrost can have up to 12.24% (*CVM+RIF*) and 15.00% (*CVM+PI*) performance improvement for applications with high TX traffic.

Combining *+ZC* and *+PRPR* in the RX direction shows a greater performance improvement than the sum of each technique's individual improvement (explained in § 7.2.2). The TX improvement is less significant because PRPR only optimizes RX traffic. Limited CPU resources on a single vCPU for both packet processing and TLS operations result in a large gap from reaching the NIC's maximum bandwidth.

7.2.2 Applications

We utilize the same network-intensive applications as in § 3 to evaluate and break down the performance improvement of Bifrost. TLS/SSL is enabled in all the applications. We run each benchmark for 30 seconds and report the average value of the results from 10 rounds. To save space, we only present and analyze the results of the 32KB and the 256KB data sizes in detail, and provide an overview of the results for other data sizes. The detailed benchmark configurations and results are shown below.

Memcached [20] is a popular multi-threaded in-memory key-value store application. We use the *memtier_benchmark* [56] tool to measure throughput and average latency. The *Memcached* server is configured with either 1 or 4 threads for VM with 1 or 4 vCPUs respectively, and 4096MB memory. We set up 4 clients, 16 concurrent requests for 1-thread server and 8 clients, 32 concurrent requests for 4-thread server.

Figure 7a and Figure 8a show the throughput and latency overhead, respectively, of Memcached in *CVM+RIF*. Both throughput and latency improve as a result of alleviating the vCPU bottleneck. In 32KB cases, Bifrost cuts down more than half of *CVM+RIF*'s overhead over its baseline. Either *+ZC* or *+PRPR* alone slightly mitigates the overhead. In 256KB cases, Bifrost performs about 10% better than its baseline. Either *+ZC* or *+PRPR* alone reduces the overhead by more than half. With the same number of vCPUs, Bifrost's performance improvement increases as the data size grows. This is primarily because the performance impact of the CVM-IO tax, especially the bounce buffer tax, becomes more pronounced with the growth of data size, providing more room for improvement.

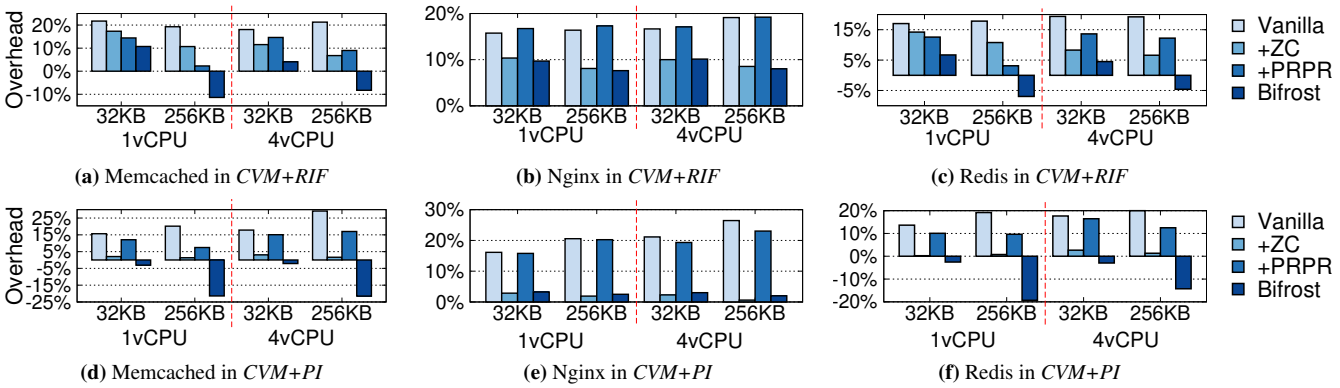
Figure 9a displays the time breakdown and backend utilization of CPUs in 4vCPU-256KB cases. The ZCED reduces the total timeshare of the bounce buffer tax from 15.67% to less than 2.50%. It cannot completely eliminate the bounce buffer cost because some small I/O data (e.g., TCP handshake packets) still falls back to the bounce buffer. The PRPR reduces the timeshare of the packet processing tax from 28.73% to 21.88%. Bifrost spends 2.39% more time on application workloads than the baseline and has more than 10% speed gain on the TLS processing in application workloads, which explains the 8.26% improvement over the baseline. Due to the higher throughput and PRPR cost, Bifrost's backend CPU utilization increases by 8.75% compared to the baseline.

Figure 7d and Figure 8b show the throughput and latency overhead, respectively, of Memcached in *CVM+PI*. Bifrost outperforms the baseline in all cases. The throughput acceleration over the baseline can reach 3.06% in 32KB cases and 21.50% in 256KB cases. The latency overhead is almost eliminated in 32KB cases and can outperform the baseline by 17.45% in 256KB cases. *+ZC* obtains more individual performance gain than *+PRPR*, and their combined improvement is larger than the sum of their individual gains. This is because applying both techniques can provide more available CPU cycles to application workloads than applying only one of them. As shown in Figure 9a, when only applying PRPR, part of the released CPU cycles will be occupied by bouncing packets.

Figure 9b depicts the breakdown and backend utilization of CPUs in 4vCPU-256KB cases. The ZCED reduces the total timeshare of the bounce buffer tax from 19.45% to less than 2.77%. The PRPR reduces the timeshare of the packet processing tax from 36.14% to 26.83%. Compared to the baseline, Bifrost provides application workloads with 11.53% more CPU time and more than 10% speedup in TLS processing. This can explain the 21.50% improvement over the baseline. Due to the higher throughput and PRPR cost, Bifrost's backend CPU utilization is 19.05% more than the baseline.

Nginx [50] is a well-known high-performance HTTP(S) web server. We run the *wrk* [21] benchmark tool to measure the throughput represented by requests per second (RPS). The client configurations are similar to the other two applications.

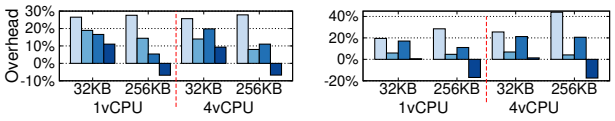
Figure 7b illustrates the Nginx throughput overhead of *CVM+RIF*. Since the traffic type of the Nginx benchmark is mainly in the TX direction, the majority of Bifrost's performance improvement comes from the ZCED, as analyzed in § 7.2.1. *+ZC* reduces the overhead by less than half because lengthy VM exits still significantly impact performance. The PRPR even increases the overhead for a little bit in the 1vCPU-256KB case, because there are more VM exits after



(a) Memcached in *CVM+RIF* **(b)** Nginx in *CVM+RIF* **(c)** Redis in *CVM+RIF*

(d) Memcached in *CVM+PI* **(e)** Nginx in *CVM+PI* **(f)** Redis in *CVM+PI*

Figure 7: Application performance comparisons when applying some or all of Bifrost's techniques. The Y-axis indicates relative overhead compared with baseline VMs, negative overhead represents performance improvement. (a), (b) and (c) compare throughput of *CVM+RIF* on the AMD server. (d), (e) and (f) compare throughput of *CVM+PI* on the Intel server.



(a) Latency in *CVM+RIF* **(b)** Latency in *CVM+PI*

Figure 8: Memcached average latency comparisons when applying some or all of Bifrost's techniques. The Y-axis indicates relative overhead compared with baseline VMs, negative overhead represents performance improvement.

the PRPR is applied. Figure 7e shows the Nginx throughput overhead of *CVM+PI*. Bifrost brings the overhead to less than 2.8% in all cases, thanks to the ZCED. The PRPR no longer impacts the performance negatively because VM exits cost is trivial when posted interrupt is enabled.

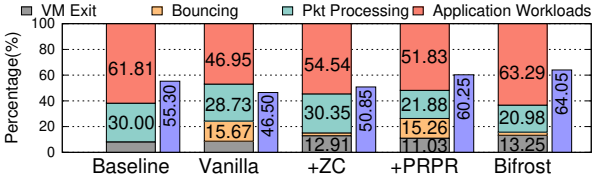
Redis [28] is a single-threaded in-memory key-value store application widely deployed in production environments. We also use the *memtier_benchmark* tool to measure the throughput. The *Redis* server is configured with 4096MB memory. The *memtier_benchmark* uses the same configurations as *Memcached*. To fully utilize vCPU resources in 4vCPU cases, we use *redis-cli* to build a *Redis* cluster with 4 instances.

Figure 7c and Figure 7f present the Redis throughput overhead, which have similar patterns to those of Memcached.

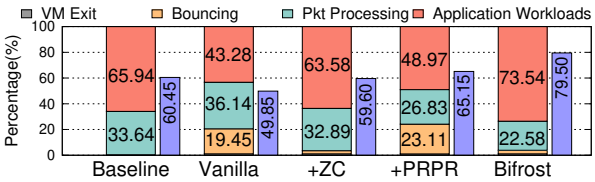
VM Scalability: To show Bifrost is scalable as the number of VM grows, we evaluate applications in 1, 2 and 4 Bifrost-enabled 4-vCPU 2-virtqueue *CVM+RIF*s. We conduct experiments on the AMD server because it has sufficient CPU cores on a single NUMA node. Figure 10 demonstrates that in multi-VM cases, Bifrost can always achieve comparable performance improvements to that of the single VM scenario. Bifrost's good VM scalability comes naturally because ZCED uses Linux's scalable memory allocator and PRPR is applied to each virtqueue without contention.

7.3 TOCTTOU Protection Overhead

Bifrost defends the guest OS against TOCTTOU attacks with OTTR by copying packet headers into private memory and keeping end-to-end encrypted payload in registers during

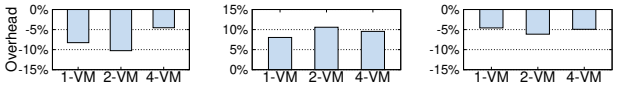


(a) Breakdown of *CVM+RIF* and its baseline & Backend CPU utilization



(b) Breakdown of *CVM+PI* and its baseline & Backend CPU utilization

Figure 9: VM CPU time breakdown and backend CPU utilization in Memcached 4vCPU-256KB experiments on AMD and Intel servers. The Y-axis represents the percentage of CPU cycles. In each case, the left side shows the breakdown of CPUs that run VMs, while the right side presents the utilization of backend CPUs.



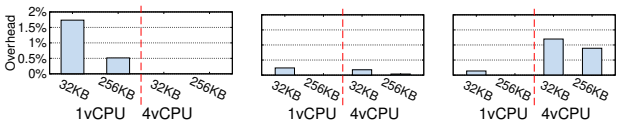
(a) Memcached **(b)** Nginx **(c)** Redis

Figure 10: Performance comparisons of Bifrost in 1, 2 and 4 *CVM+RIF* VMs using Memcached 4vCPU-256KB experiments. The Y-axis indicates relative overhead compared with baseline VMs, negative overhead represents performance improvement.

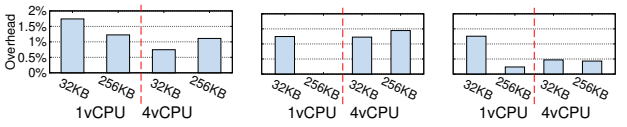
their processing. To evaluate the performance impact of these operations, we implement a prototype of Bifrost without applying OTTR, called Bifrost-noprot. We repeat application benchmarks to compare Bifrost's performance with that of Bifrost-noprot. The overhead of Bifrost in different benchmarks is shown in Figure 11, indicating no more than 2.0% overhead caused by TOCTTOU protections in all cases.

7.4 Memory Footprint

Bifrost must utilize memory efficiently to avoid unavailability due to the depletion of the ZCED NUMA memory. We first



(a) Memcached in *CVM+RIF* **(b)** Nginx in *CVM+RIF* **(c)** Redis in *CVM+RIF*



(d) Memcached in *CVM+PI* **(e)** Nginx in *CVM+PI* **(f)** Redis in *CVM+PI*

Figure 11: Application performance overhead of TOCTTOU protection. (a), (b) and (c) compare Bifrost with Bifrost-noprot (i.e., w/o OTTR) on the AMD server. (d), (e) and (f) compare Bifrost with Bifrost-noprot on the Intel server.

measure the ZCED NUMA's memory consumption via the *numastat* tool. Among all of our benchmarks, Bifrost consumes no more than 200MB out of the 512MB capacity of the ZCED NUMA memory. This amount does not exceed the default 1GB size of the bounce buffer area in the 16GB *CVM+RIF* VM. As for the footprint in the network backend, PRPR maintains a packet cache list in the backend memory for each virtqueue. Each cache list contains at most 8 network flows, each caching at most 1024 network packets. The maximum memory cost of one cache list is 1,088KB. In our benchmarks, we enable 2 virtqueues, consuming only 2.125MB memory.

8 Security Analysis

Bifrost introduces three major techniques to existing CVMs, in which only the ZCED and the OTTR retrofit the guest kernel and may have an impact on the network I/O data security. The only difference between the network I/O of Bifrost and a vanilla CVM is that Bifrost needs to process packets in the guest-host shared memory, while a vanilla CVM handles packets in private memory. Thus, we only need to analyze the security risks caused by TOCTTOU attacks on network packets during network I/O.

Headers: In the RX direction, a header is received in the guest-host shared memory. Bifrost copies the header into private memory, and subsequent header processing only uses the private copy, which does not suffer from TOCTTOU attacks. In the TX direction, each header is born in private memory and sent out through the bounce buffer mechanism, which is the same as in the existing CVMs.

Encrypted payload: In the RX direction, the in-kernel TLS layer decrypts the encrypted payload from the guest-host shared memory into private memory. Bifrost ensures that the decryption code has a consistent view of the encrypted payload by reading from shared memory only once and keeping it in CPU registers. In the TX direction, the in-kernel TLS layer encrypts the plaintext payload directly into shared memory. Bifrost ensures that the encryption code always refers to the correct ciphertext in CPU registers, which is isolated by CVM platforms and immune to TOCTTOU attacks.

Plaintext payload: There are also packets carrying plaintext payload due to procedures such as handshaking. In the RX direction, the plaintext payload is not accessed until the guest kernel copies it from shared memory to private memory. In the TX direction, the plaintext payload is no longer accessed once the guest kernel copies it from private memory to shared memory. Avoiding shared memory access eliminates the risks of TOCTTOU vulnerability.

Therefore, Bifrost does not expose guest OS's network processing to TOCTTOU attacks, achieving the same level of security guarantees as vanilla CVMs.

9 Related Work

Secure Virtualized Systems. A long line of research works and commercial products have been proposed to build secure virtualized systems [3, 8, 10, 11, 26, 32, 38, 48]. AMD SEV [2, 3], Intel TDX [29, 32] and ARM CCA [8] enable the CVM abstraction with hardware extensions, especially memory encryption and integrity protection [30, 34]. While AMD SEV relies on a secure processor [1], Intel TDX and ARM CCA employ trusted firmware [31] to manage CVMs. TwinVisor [38] provides an TrustZone-based alternative to ARM CCA by retrofitting the virtualization extension on existing ARM platforms. The design of Bifrost is not restricted to AMD or Intel and can be applied to other CVM systems.

Zero Copy I/O. Prior research works have proposed various techniques to eliminate data copies for better I/O performance [24,27,35,42–44]. For user-level applications, zIO [58] can transparently remove redundant I/O copies. For the I/O stack in the kernel, DAMN [44] and Demikernel [63] eliminate I/O memory copies by directly allocating buffers from the I/O memory pool. PASTE [25] performs DMA directly into non-volatile memory to avoid copies. Unlike these systems that target traditional scenarios and/or require intrusive software modifications, Bifrost focuses on eliminating unnecessary I/O data copies in CVMs with minor modifications.

10 Conclusion

This paper presents the first systematic analysis of the CVM-IO tax for network-intensive workloads in CVMs. To optimize the I/O performance of CVMs, we propose a new paravirtual I/O design called Bifrost. Bifrost eliminates redundant packet bounces and greatly reduces packet processing cost, while maintaining the same level of security guarantees as existing CVMs. Evaluation results show that Bifrost significantly improves the I/O performance of CVMs, and even outperforms traditional VMs by up to 21.50%.

11 Acknowledgments

We express our sincere gratitude to our shepherd Kartik Gopalan, whose valuable suggestions have greatly improved our paper. We thank the anonymous reviewers for their insightful suggestions. This work was partially supported by NSFC (No. 62002218 and 61925206), the Fundamental Research Funds for the Central Universities, NSFC (No. 62132014 and 62141218) and Huawei Innovation Research Plan.

References

[1] BlackHat 2020. All you ever wanted to know about the AMD Platform Security Processor and were afraid to emulate. https://i.blackhat.com/USA-20/Wednesday/us-20-Buhren-All-You-Ever-Wanted-To-Know-About-The-AMD-Platform-Security-Processor-And-Were-Afraid-To-Emulate.pdf, 2020.

[2] AMD. Protecting VM Register State With SEV-ES. https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf, 2017.

[3] AMD. 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, 2020.

[4] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf, 2022.

[5] AMD. AMD64 Architecture Programmer's Manual Volume 2. https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1021, 2023.

[6] BEN ARENT. Securing MySQL Databases with SSL/TLS. https://goteleport.com/blog/secure-database-with-tls/, 2022.

[7] ARM. What are key features in GICv3.x and GICv4.x? https://developer.arm.com/documentation/ka004701/latest, 2022.

[8] ARM. ARM Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2023.

[9] Microsoft Azure. Azure AI. https://azure.microsoft.com/en-us/solutions/ai/#overview, 2023.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[11] Jiahao Chen, Dingji Li, Zeyu Mi, Yuxuan Liu, Binyu Zang, Haibing Guan, and Haibo Chen. Security and Performance in the Delegated User-level Virtualization. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA, July 2023. USENIX Association.

[12] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 601–608, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Google Cloud. Memorystore for Memcached overview. https://cloud.google.com/memorystore/docs/memcached/memcached-overview, 2023.

[14] Google Cloud. Memorystore for Redis overview. https://cloud.google.com/memorystore/docs/redis/redis-overview, 2023.

[15] Jonathan Corbet. JLS2009: Generic receive offload. *Linux Weekly News (LWN)*, 2009.

[16] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc De Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 373–387, USA, 2018. USENIX Association.

[17] Docker. Protect the Docker daemon socket. https://docs.docker.com/engine/security/protect-access/, 2022.

[18] Morris J Dworkin. *NIST Special Publication 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. National Institute of Standards & Technology, 2007.

[19] Jake Edge. TLS in the kernel. https://lwn.net/Articles/666509/, 2015.

[20] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[21] Will Glozer. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk, 2022.

[22] Google. Confidential Space security overview. https://cloud.google.com/docs/security/confidential-space, 2022.

[23] Leonid Grossman. Large receive offload implementation in neterion 10GbE Ethernet driver. In *Linux Symposium*, page 195, 2005.

[24] P. Halvorsen, E. Jorde, K.-A. Skevik, V. Goebel, and T. Plagemann. Performance tradeoffs for static allocation of zero-copy buffers. In *Proceedings of the 28th Euromicro Conference*, pages 138–143, 2002.

[25] 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*, NSDI'18, page 17–33, USA, 2018. USENIX Association.

[26] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. Confidential Computing for OpenPOWER. In *Proceedings of the 16th European Conference on Computer Systems*, EuroSys '21, page 294–310, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, April 2014. USENIX Association.

[28] Redis Inc. Introduction to Redis. https://redis.io/docs/about/, 2022.

[29] Intel. Intel TDX® Module v1.5 Base Architecture Specification. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-module-1.5-base-spec-348549001.pdf, 2022.

[30] Intel. Intel® Architecture Memory Encryption Technologies. https://www.intel.com/content/www/us/en/develop/download/intel-mktme-specification.html, 2022.

[31] Intel. Intel® Trust Domain Extension (Intel® TDX) Module. https://www.intel.com/content/www/us/en/download/738875/intel-trust-domain-extension-intel-tdx-module.html, 2022.

[32] Intel. Intel® Trust Domain Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf, 2022.

[33] Li Jie, Chen Shuhui, and Su Jinshu. Implementation of TCP large receive offload on multi-core NPU platform. In *2016 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 258–263, 2016.

[34] David Kaplan. AMD x86 Memory Encryption Technologies. Austin, TX, August 2016. USENIX Association.

[35] Yousef A. Khalidi and Moti N. Thadani. An Efficient Zero-Copy I/O Framework for UNIX. Technical report, Sun Microsystems, Inc., USA, 1995.

[36] Christoph Lameter. Local and remote memory: Memory in a Linux/NUMA system. In *Linux symposium*, pages 1–25, 2006.

[37] Christoph Lameter. NUMA (Non-Uniform Memory Access): An Overview: NUMA Becomes More Common Because Memory Controllers Get Close to Execution Units on Microprocessors. *Queue*, 11(7):40–51, jul 2013.

[38] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. TwinVisor: Hardware-Isolated Confidential Virtual Machines for ARM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 638–654, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 337–351, 2022.

[40] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CrossLine: Breaking "Security-by-Crash" Based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2937–2950, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference*, ACSAC '21, page 609–619, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2010.

[43] 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*, ASPLOS '16, page 249–262, New York, NY, USA, 2016. Association for Computing Machinery.

[44] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. DAMN: Overhead-Free IOMMU Protection for Networking. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 301–315, New York, NY, USA, 2018. Association for Computing Machinery.

[45] 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.

[46] Eugenio Pérez Martín and Adrian Moreno Zapata. A journey to the vhost-users realm. https://www.redhat.com/en/blog/journey-vhost-users-realm, 2019.

[47] Zeyu Mi, Haibo Chen, Yinqian Zhang, Shuanghe Peng, Xiaofeng Wang, and Michael K. Reiter. CPU Elasticity to Mitigate Cross-VM Runtime Monitoring. *IEEE Transactions on Dependable and Secure Computing*, 17(5):1094–1108, 2020.

[48] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. (Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, pages 1695–1712. USENIX Association, August 2020.

[49] MongoDB. Configure mongod and mongos for TLS/SSL. https://www.mongodb.com/docs/manual/tutorial/configure-ssl/, 2022.

[50] Nginx. Nginx. https://www.nginx.com/, 2022.

[51] NVIDIA. ConnectX SmartNICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/, 2022.

[52] Hitoshi Oi and Fumio Nakajima. Performance Analysis of Large Receive Offload in a Xen Virtualized System. In *2009 International Conference on Computer Engineering and Technology*, volume 1, pages 475–480, 2009.

[53] Michele Paolino, Nikolay Nikolaev, Jeremy Fanguede, and Daniel Raho. SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 86–92, 2015.

[54] Joana Pecholt and Sascha Wessel. CoCoTPM: Trusted Platform Modules for Virtual Machines in Confidential Computing Environments. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 989–998, New York, NY, USA, 2022. Association for Computing Machinery.

[55] PostgreSQL. Secure TCP/IP Connections with SSL. https://www.postgresql.org/docs/current/ssl-tcp.html, 2022.

[56] Redis. memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached. *https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/*, 2013.

[57] Amazon Web Services. AWS for Financial Services. https://aws.amazon.com/financial-services, 2023.

[58] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 431–445, Carlsbad, CA, July 2022. USENIX Association.

[59] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzicker Chiueh. A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, page 1–15, New York, NY, USA, 2015. Association for Computing Machinery.

[60] Wikipedia. Terabit Ethernet. https://en.wikipedia.org/wiki/Terabit_Ethernet, 2022.

[61] Dan York. Google Is Now Always Using TLS/SSL for Gmail Connections. https://www.internetsociety.org/blog/2014/03/google-is-now-always-using-tlsssl-for-gmail-connections/, 2014.

[62] Dongli Zhang. swiotlb: 64-bit DMA buffer. https://lwn.net/Articles/845096/, 2021.

[63] 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.

SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning

Junming Ma, Yancheng Zheng,* Jun Feng, Derun Zhao, Haoqi Wu
Wenjing Fang, Jin Tan,† Chaofan Yu, Benyu Zhang, Lei Wang

Ant Group

Abstract

With the increasing public attention to data security and privacy protection, privacy-preserving machine learning (PPML) has become a research hotspot in recent years. Secure multi-party computation (MPC) that allows multiple parties to jointly compute a function without leaking sensitive data provides a feasible solution to PPML. However, developing efficient PPML programs with MPC techniques is a great challenge for users without cryptography backgrounds.

Existing solutions require users to make efforts to port machine learning (ML) programs by mechanically replacing APIs with PPML versions or rewriting the entire program. Different from the existing works, we propose SecretFlow-SPU, a performant and user-friendly PPML framework compatible with existing ML programs. SecretFlow-SPU consists of a frontend compiler and a backend runtime. The frontend compiler accepts an ML program as input and converts it into an MPC-specific intermediate representation. After a series of delicate code optimizations, programs will be executed by a performant backend runtime as MPC protocols. Based on SecretFlow-SPU, we can run ML programs of different frameworks with minor modifications in a privacy-preserving manner.

We evaluate SecretFlow-SPU with state-of-the-art MPC-enabled PPML frameworks on a series of ML training tasks. SecretFlow-SPU outperforms these works for almost all experimental settings (23 out of 24). Especially under the wide area network, SecretFlow-SPU is up to 4.1× faster than MP-SPDZ and up to 2.3× faster than TF Encrypted.

1 Introduction

Privacy-preserving machine learning (PPML) [24, 27, 34, 43, 44, 47, 49, 56, 57] has been gaining popularity due to the pervasive usage of machine learning (ML) and attendant privacy problems. Secure multi-party computation (MPC) [39], a cryptographic technique that enables multiple parties to jointly compute a function without leaking each party's private inputs, brings a provable and practical solution to ML users with strong privacy concerns. For example, financial and medical data analysts can collaboratively train a model on private datasets that contain sensitive information.

However, incorporating MPC techniques in ML applications introduces great challenges due to the natural differences between these two fields. MPC experts mainly focus on designing performant cryptographic protocols for low-level computation primitives. In contrast, ML practitioners are more accustomed to constructing high-level ML models using user-friendly frameworks that encapsulates commonly-used ML building blocks. Consequently, it poses a massive obstacle for ML users without cryptography expertise to achieve complex PPML tasks efficiently in real-world scenarios.

A series of works have been proposed to eliminate this obstacle. EzPc [9], ABY [15], MP-SPDZ [29], etc. [18] design domain-specific languages (or use high-level languages) to provide general purpose MPC compilers and support arbitrary computations upon MPC. These works significantly reduce the difficulty of developing MPC programs and allow for MPC-specific compilation optimizations. Whereas, these works remain a significant gap from mainstream ML frameworks on API designs, thus lacking user-friendliness to develop complex ML programs.

TF Encrypted [14] and CrypTen [33] take a step further in this direction by providing general ML interfaces with MPC implementations. These works mimic the existing ML frameworks' API designs (e.g., TensorFlow [5] and PyTorch [45]) to hide the underlying MPC cryptographic details and gain further user-friendliness. However, efforts still need to be made to port ML programs from TensorFlow/PyTorch by substituting PPML version APIs mechanically. Take CrypTen as an example: given a pre-defined PyTorch model, the user must manually re-write the model training/prediction programs by replacing PyTorch tensors, loss function, and optimizer with CrypTen counterparts. Besides, these frameworks rely on TensorFlow or PyTorch as their underlying runtime, which lacks

*Junming and Yancheng contribute equally in this work.
†Corresponding author: tanjin.tj@antgroup.com.

MPC domain-specific knowledge for compilation optimizations.

A question then arises: *can we efficiently run ML programs of mainstream frameworks in a privacy-preserving manner?* As an attempt to answer this question, we propose SecretFlow-SPU in this paper. For simplicity, we will refer to it as SPU throughout the rest of this paper. SPU is a general-purpose PPML framework designed to bridge the gap between ML and MPC communities more naturally. The core components of SPU include a frontend compiler and a backend MPC runtime. SPU provides users with Python APIs to accept an ML program (with minor modifications to specify protected data) as input, and the frontend compiler emits a customized intermediate representation (IR) named PPHLO (short for **P**rivacy-**P**reserving **H**igh-**L**evel **O**perations) as output. The backend runtime is a virtual device built on multiple connected computing nodes, which receives PPHLO and executes it as MPC protocol implementations among nodes to complete private ML training or prediction.

SPU's architecture makes it friendly to both ML and MPC developers. On the one hand, ML developers can conveniently run ML applications developed through mainstream ML frameworks in a privacy-preserving manner on SPU (Section 3.3) without the cryptographic knowledge of MPC. Besides, SPU is not bound to one specific ML framework. Diverse frameworks and libraries can be supported in SPU if there is a path from ML source code to PPHLO. On the other hand, SPU provides great extensibility to MPC protocol developers, who only need to focus on designing fundamental MPC primitives and implementing corresponding APIs defined by SPU. New MPC protocol supports can be easily supplemented without caring about high-level ML workflows (Section 3.6.1).

Besides user-friendliness, PPHLO enables us to propose and implement MPC-specific optimizations at both frontend and backend to achieve high performance. At the frontend, we observe that traditional ML frameworks usually generate a computation graph that is not optimal for MPC computations. The reason behind the observation is that MPC computations have a rather different cost model than plaintext computations due to additional communication overhead. Based on PPHLO, we design and implement several compiler passes to generate more efficient IR (Section 3.5). At the backend, we implement strategies such as vectorization and streaming to reduce MPC communication overhead. Meanwhile, SPU backend runtime employs inter- and intra-operation concurrency to execute PPHLO efficiently (Section 3.6.2).

We develop SPU frontend and backend in C++ and provide PPML developers with Python APIs to run applications. We mainly use ML programs written in JAX [8] to evaluate SPU's performance and user-friendliness. For performance, we use three state-of-the-art MPC-enabled PPML frameworks (i.e., MP-SPDZ [29], TF Encrypted [14], and CrypTen [33]) as the baseline. We train four common-evaluated neural networks on the MNIST [37] dataset for image classification under both local area network (LAN) and wide area network (WAN) settings. SPU achieves comparable classification accuracy and superior training speed. Concretely, SPU outperforms the state-of-the-art works for almost all the settings (23 out of 24). Especially under the WAN setting, SPU is up to 4.1× faster than MP-SPDZ and up to 2.3× faster than TF Encrypted.

Regarding user-friendliness, we evaluate SPU by running JAX programs from popular open-source JAX projects' official examples. We only need to modify a few lines of code to make these examples run seamlessly on SPU. Experimental results show that SPU can be easily applied to other models such as Long Short-Term Memory [23] and Variational Auto-Encoder [32]. This compatibility is hard to achieve for existing MPC-enabled frameworks. Moreover, we also validate SPU's feasibility in supporting different ML frameworks by running TensorFlow and PyTorch programs.

The contributions we make in this paper are summarized as follows:

- We design and implement SPU as the first MPC-enabled PPML framework to support ML programs (with minor modifications) from different mainstream ML frameworks, significantly accelerating the development, testing, debugging, and deployment of PPML applications.
- We design an MPC-specific IR, i.e., PPHLO, which connects ML and MPC worlds. Besides, we propose/implement a series of compilation optimizations and develop a high-performance runtime to execute PPHLO.
- We validate SPU with a series of experiments on performance and user-friendliness. The experimental results demonstrate SPU's efficiency and ease of use.
- We open-source SPU to bolster the advancement of PPML for academic and industry communities. The code is available at https://github.com/secretflow/spu.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to the background of ML compilers and MPC. We describe the design details of SPU in Section 3. Section 4 describes the system implementation and evaluation results. Section 5 describes SPU's limitations. Related work is discussed in Section 6. Finally, we conclude this paper in Section 7.

2 Background

SPU is an interdisciplinary work of ML compiler and MPC. In order to better understand the motivation and design of SPU, we give a more detailed description of the background of ML compiler and MPC in this section.

2.1 Machine Learning Compilers

In the past decade, artificial intelligence technologies driven by ML have made numerous breakthroughs in many fields, such as natural language processing [55], computer vision [19], and drug discovery [26]. As the key infrastructure for implementing ML algorithms, an easy-to-use and efficient ML framework is crucial. A large number of ML frameworks (and libraries) are currently on the market, including TensorFlow [5], PyTorch [45], JAX [8], and MxNet [11], providing developers with similar capabilities to train and serve models.

Meanwhile, in addition to traditional CPU and GPU, many application-specific integrated circuits for ML workloads have been developed to accelerate program execution. Typical representatives are Google TPU [25] and Hisilicon NPU [38]. Generating machine code for different ML frameworks to adapt to different types of hardware requires substantial engineering efforts, especially when the number of ML frameworks and hardware devices keeps increasing. ML compilers are proposed as the solution to this problem. Usually, the compiler frontend will transform source code written with the existing frameworks into hardware-independent IR, and the compiler backend will further transform IR into hardware-dependent machine code. With ML compilers, frontend frameworks only need to focus on generating IR, and backend hardware vendors only need to pay attention to supporting IR instructions.

The IR used in ML compilers is typically expressed as a computation graph (i.e., a directed acyclic graph). Graph nodes are ML operations (such as matrix multiplication and convolution) whose input and output are tensors (i.e., multi-dimensional arrays). Graph edges show the data dependencies between operations. One widely-used ML compiler is Google's XLA [4]. XLA defines its IR as HLO (High-Level Operations) to represent computation graphs. A series of frontends, including TensorFlow, PyTorch, and JAX, support XLA. ML programs written in these frameworks can be compiled into HLO. After performing hardware-independent and hardware-dependent optimizations, HLO is finally lowered to machine code by the XLA backend to run on the CPU, GPU, or TPU.

2.2 Secure Multiparty Computation

MPC originates from the Yao's Millionaires' problem [59] in the 1980s, where two rich people want to compare their wealth without giving away the exact value. Beyond this, MPC has shifted from an academic theory to practical usage in more complicated tasks, such as training ML models [30, 44, 56].

One fundamental technique used in MPC is secret sharing [6, 51]. A secret value is divided into multiple random shares and distributed to several parties. Each party only gets a subset of the shares and cannot reconstruct the original value independently. These parties jointly compute pre-defined computations (e.g., ML training) without leaking any sensitive information of the inputs or intermediate computation results. Usually, the final computation result (e.g., the trained model) are revealed to some designated parties. At that time, all parties put together their holding shares to reconstruct the result.

The private inputs, including integer and boolean values, are typically encoded over an algebraic ring or finite field. For integers, arithmetic secret sharing encrypts a secret over the ring $\mathbb{Z}_{2^k}$ and supports efficient arithmetic operations, including additions and multiplications. Correspondingly for boolean values, binary secret sharing provides a scheme to encrypt a secret over the ring $\mathbb{Z}_2$ and supports more efficient boolean operations, including XOR and AND computations. Addition (resp., XOR) operation of two arithmetic (resp., boolean) secret shares is equal to add (resp., XOR) the share of the two secrets locally. Operations with similar local-computation properties include a secret value adding a public value and a secret value multiplying a public value. In contrast, the multiplication (resp., AND) of two secret values is more complicated, which requires additional communication among the participating parties to exchange extra information. The heavy communication overhead has weakened the performance of MPC, especially in handling complex computations in real-world scenarios.

To improve the efficiency, mixed-protocols [15, 43, 46] that use arithmetic and binary secret sharing interchangeably shed light on MPC. With dedicated protocols, arithmetic and binary secret shares can be transformed back and forth to handle complex computations, including both arithmetic and non-arithmetic computations. However, these conversions also need communication. Despite the great efforts that have been made, the performance of MPC operations is still heavily communication-bound and sensitive to the network environment. Such characteristic makes MPC significantly different from traditional plaintext computations over CPU.

Besides integer and boolean operations, MPC also supports decimal computations, which are common in ML. It is more common and efficient to encode decimals as fixed-point numbers, which can be interpreted as the integer value multiplying a scaling factor. This factor is configurable and indicates how many bits represent the fractional part, and the remaining bits (except the sign bit) represent the integer part. When a fixed-pointed value multiplies another fixed-pointed value, the fractional bits double. In order to maintain that the result has consistent fractional bits with the input, a truncation operation is required.

The addition and multiplication of integers, fixed-point numbers, and boolean logic operations constitute MPC's most fundamental building blocks. ML scenarios require more complex and high-level operations. For linear operations such as matrix multiplication and convolution, a combination of those basic building blocks can accomplish them. For non-linear operations such as activation functions, we can approximate these functions using mathematical algorithms like Newton-

Raphson method [60]. Based on the linear and non-linear operations, complex ML tasks can be completed.

3 System Design

We describe the detailed system design of SPU in this section. The threat model of SPU is given first, followed by an overview of SPU's architecture and individual descriptions of SPU's main components, including programming interfaces, PPHLO, frontend, and backend.

3.1 Threat Model

SPU follows a standard MPC threat model and runs predefined programs among multiple parties, protecting input data and all intermediate results, typically only revealing the final results to some designated parties. Taking ML model training as an example, MPC can protect participating parties' training datasets and intermediate results like gradients, and reveal the trained model weights as the final results.

Furthermore, as an MPC computing engine, SPU is not restricted to any specific MPC threat models, such as the number of participating parties or if participants behave honestly [39]. The underlying MPC protocol used in SPU is configurable, allowing the threat model of the entire SPU system to be determined by the selected MPC protocol at runtime. For instance, using the semi-honest (with honest majority) ABY3 [43] protocol in SPU indicates SPU inherits ABY3's threat model, i.e., all participants follow the protocol honestly but may attempt to gain additional information from exchanged messages.

3.2 Architecture Overview

The goal of SPU is to run ML programs in a privacy-preserving manner. To complete this goal, we propose SPU's architecture, as illustrated in Figure 1. In the rest of this section, we use JAX as an example ML framework to describe SPU's design although SPU is not limited to JAX. We give evaluations on SPU's support to other frameworks in Section 4.2.3. Given an ML program written in JAX, our programming interfaces will implicitly call JAX API to convert this program into HLO (Section 2.1). This HLO graph and data visibility defined by users will be passed to SPU frontend, which compiles HLO to SPU's customized IR, i.e., PPHLO. After generating PPHLO, the frontend will further perform MPC-specific optimizations. The optimized PPHLO will then be sent to SPU backend, a virtual device built on multiple networked computing nodes. These nodes host SPU runtime responsible for executing MPC operations, and their number should match the supported parties of the configured MPC protocol.

SPU employs the SPMD (Single-Program-Multiple-Data) programming model. All nodes receive the same PPHLO to execute. The data consumed by each node are secret shares

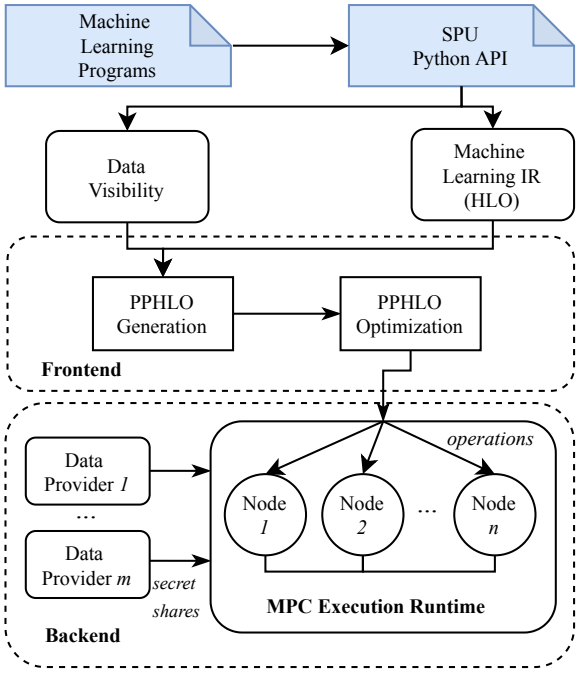


Figure 1: SPU architecture.

derived from original data held by data providers. Following the workflow described above, sensitive data from different sources can be jointly used to finish a PPML job by simply running a JAX program on SPU.

The design of SPU has the following advantages:

- By supporting ML programs of mainstream ML frameworks, SPU is extremely easy to use. SPU does not require users to learn a new library or language, or have MPC expertise.
- SPU frontend consumes HLO generated by ML programs rather than the source code. This design choice makes SPU can support a series of existing frameworks that have a lowering path to HLO directly. Moreover, SPU can benefits from platform-independent optimizations from existing ML compilers.
- SPU backend is also extensible. We can implement multiple pluggable MPC protocols in the backend without modifying PPHLO and frontend programs.
- PPHLO allows SPU to do systematic optimizations at the granularity of the high-level computational graph, which enables it to generate high-performance code for MPC execution.

3.3 Programming Interface

In order to achieve ease of use, we provide simple Python APIs so that developers can run ML programs on SPU with

```
1  import jax.numpy as jnp
2  import numpy as np
3  import spu.binding.util.distributed as ppd
4
5  # init SPU backend nodes
6  with open("/path/to/config", 'r') as file:
7      conf = json.load(file)
8  ppd.init(conf["nodes"], conf["devices"])
9
10 # specify data visibility
11 @ppd.device("P1")
12 def data_from_alice():
13     return np.random.randint(100, size=(4,))
14
15 # specify data visibility
16 @ppd.device("P2")
17 def data_from_bob():
18     return np.random.randint(100, size=(4,))
19
20 # specify a private function
21 @ppd.device("SPU")
22 def compare(x, y):
23     return jnp.maximum(x, y)
24
25 # x & y will be automatically
26 # fetched by SPU (as secret shares)
27 x = data_from_alice()
28 y = data_from_bob()
29
30 # compare will be evaluated privately by SPU
31 z = compare(x, y)
32
33 # reveal the real value of z
34 print(f"z = {ppd.get(z)}")
```

Figure 2: A demonstration of how to use SPU's API run JAX programs privately. Developers use decorators to specify data visibility and functions to be protected.

a few lines of code modifications. An example is given in Figure 2. We use SPU to solve Yao's Millionaires' problem as a simple demonstration.

We assume the two participants are called Alice and Bob. In line 3, at the start of this code, we import SPU's APIs as module *ppd*. In lines 6 to 8, we initialize the backend SPU nodes and data providers (i.e., P1 and P2 are to represent Alice and Bob). The decorators in lines 11 and 16 are data visibility marks that specify these data come from P1 and P2, which means that the two functions can only be evaluated locally on P1 and P2. The derived results are private data to be protected on SPU. The decorator on line 21 is a private function mark that specifies the function *compare* is private and should be evaluated on SPU. Lines 27 to 31 compare Alice and Bob's data. Variables x and y will be automatically fetched by SPU as secret shares, and the compared result z is also secret shares. To get the plaintext result of z, developers should use *ppd.get()* to reconstruct z as shown in line 34.

As we can see, the most crucial part of SPU's APIs is the decorator *@ppd.device()*, which is used to specify protected data and private functions. In the example demonstrated in Figure 2, the private function is a JAX *maximum* function. In fact, this can be extended to more complex JAX functions, such as an ML model training function from JAX libraries. Decorator *@ppd.device()* is the entry point for using SPU as the workflow described in Section 3.2, which will trigger HLO generation, compilation to PPHLO, and PPHLO execution. We can have SPU do all the stuff in the background by putting the decorator on top of a JAX function.

3.4 Privacy-Preserving High-Level Operations

We design PPHLO based on HLO as a customized IR for SPU because HLO lacks MPC-related semantics for optimization and efficient execution. In general, PPHLO represents a computational graph consisting of a series of operations. Each operation's input and output are tensors. The tensor type system is the most significant difference between PPHLO and other ML counterparts. A tensor's type in PPHLO can be represented by a triple <*Shape, Data Type, Visibility*>. Shape is a tensor's dimensionality. As for data type, PPHLO currently supports boolean, integer, and fixed-point numbers. Visibility is a unique tensor attribute in PPHLO. It can be either secret or public. Secret means that the tensor needs to be protected, and its real value is not visible to any node in SPU backend nodes. In contrast, public means that the tensor does not need to be protected, and any backend node can get its value.

Application developers specify the visibility of PPHLO's initial input tensors. As we described in Section 3.3, the variables generated by functions with decorator *@ppd.device()* are secret tensors, such as x and y in Figure 2. Otherwise, variables are public tensors. For each operation in PPHLO, we use the following rules to determine the output's type according to the input's type (shape is not considered here as it is determined by operation semantics). 1) **Data Type Promotion:** if one of the operands is a fixed-point number, the result is also a fixed-point number; 2) **Visibility Narrowing:** if one of the operands is a secret, the result is also a secret. Based on the two rules, we can deduce the types of all tensors in PPHLO.

Figure 3 gives an example of PPHLO in static single assignment form [50]. This code snippet corresponds to the JAX *maximum* function in Figure 2. The symbol *@main* is the program entry point. Lines 1 and 2 represent the program has two input arguments and one return value. The symbol *tensor<4x!pphlo.sec<i32>>* describes a tensor whose shape is 4, and that is a 32-bit integer secret value. Inside the braces is the program body, which contains two operations, i.e., *pphlo.greater* and *pphlo.select* (lines 3 to 6). An operation's output is assigned to the symbol on the left, which can be used as the operand of subsequent operations. When all operations

```
1  func.func @main(%arg0: tensor<4x!pphlo.sec<i32>>, %arg1: tensor<4x!pphlo.sec<i32>>)
2        -> tensor<4x!pphlo.sec<i32>> {
3    %0 = "pphlo.greater"(%arg0, %arg1) : (tensor<4x!pphlo.sec<i32>>, tensor<4x!pphlo.sec<i32>>)
4        -> tensor<4x!pphlo.sec<i1>>
5    %1 = "pphlo.select"(%0, %arg0, %arg1) : (tensor<4x!pphlo.sec<i1>>, tensor<4x!pphlo.sec<i32>>,
6        tensor<4x!pphlo.sec<i32>>) -> tensor<4x!pphlo.sec<i32>>
7    return %1 : tensor<4x!pphlo.sec<i32>>
8  }
```

Figure 3: An example of PPHLO. Generated from JAX *maximum* function shown in Figure 2.

are finished, the return value is given at line 7. PPHLO operations are extended from HLO operations [3][1]. An operation can accept public or secret tensors as its operands and has corresponding plaintext or MPC computation implementations on SPU backend runtime.

3.5 Frontend

SPU frontend is responsible for PPHLO generation and optimization. The frontend first receives an ML program's HLO and initial data visibilities as inputs and applies rules proposed in Section 3.4 to deduce the entire graph's data types and visibilities. After this step, a legal PPHLO representation is generated. The frontend will further perform code optimizations to modify PPHLO. PPHLO optimizations come from this insight: *an ML computational graph generated for non-MPC hardware may not be optimal in the MPC scenario.* We propose/implement the following compilation optimizations based on analyzing initially-generated PPHLO and our MPC expertise.

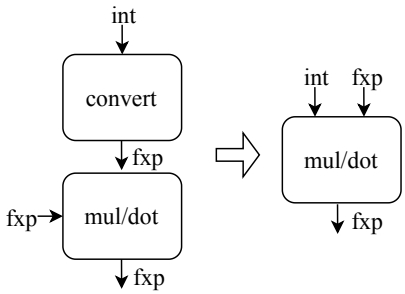


Figure 4: Mixed-data-type multiplication fusion.

Mixed-data-type multiplication fusion. In regular ML computations, when multiplying an integer to a decimal number, a *convert* operation will be called first to convert this integer to a floating-point number. Then the *multiply* operation can be dispatched to the floating-point multiplication kernel. A computation graph is illustrated in Figure 4. If we use this graph directly in SPU, an integer will be converted to a fixed-point number first, followed by a fixed-point multiplication which requires a truncation to maintain fractional bits (Section 2.2). However, an integer can directly multiply with a fixed-point number. Therefore, we can fuse the two operations into one *multiply* operation to reduce redundant truncation and conversion. This optimization applies to other similar operations like *dot*.

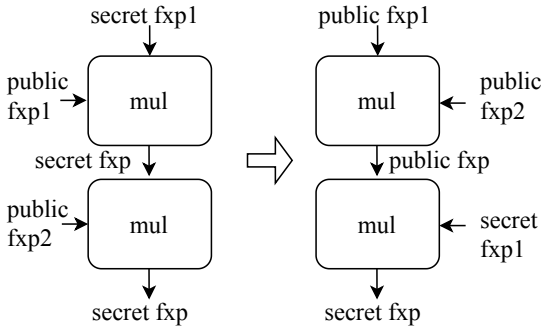


Figure 5: Mixed-visibility multiplication operands reorder.

Mixed-visibility multiplication operands reorder. Another scenario where truncation can be optimized is multiplying consecutive fixed-point numbers with mixed visibilities. As shown in Figure 5, a secret fixed-point number multiplying two public fixed-point numbers involves two *multiplication* operations. Each operation generates a secret product requiring a truncation that have a high communication overhead under some MPC protocols [24, 43]. However, we can reorder the operands without affecting the correctness. The multiplication of two public fixed-point numbers can be calculated first. The product is also public, so we can truncate the result by shifting bits locally. Then the result is used to multiply the secret fixed-point number. By reordering multiplication operands, one expensive truncation can be saved.

Inverse square root transformation. This optimization is demonstrated in Figure 6. When SPU frontend detects a computation of $y/(\sqrt{x}+u)$ where u is a tiny constant to prevent a division-by-zero problem, it will transform the computation to $y*rsqrt(x+eps())$. In the transformed computation, *eps* is

[1] Supported PPHLO operations can be found at https://github.com/secretflow/spu/blob/main/docs/reference/pphlo_op_doc.md

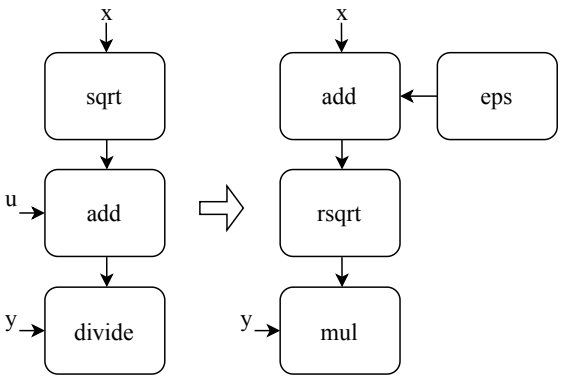


Figure 6: Inverse square root transformation.

a unique operation that will generate a minimum fixed-point number. The reason behind this transformation is that the inverse square root *rsqrt* operation has a fast MPC implementation than computing the reciprocal of the *sqrt* result (an approximation is needed). This computation pattern is observed in state-of-the-art optimizers such as Adam [31] and AMSGrad [48] when they update weights in each learning step. This optimization technique was first used by Lu et al. [41] and applied in related systems like MP-SPDZ [30] and TF Encrypted [14]. These frameworks must re-implement customized Adam and AMSGrad optimizers to employ this optimization. However, as we do the optimization at the PPHLO level, original Adam and AMSGrad optimizers in the existing JAX libraries can be directly reused.

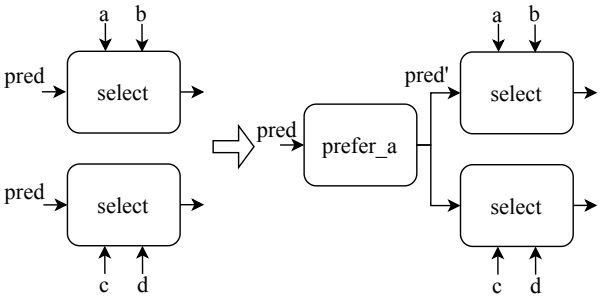


Figure 7: Select predicate reuse.

Select predicate reuse. *Select* is a commonly-used ML operation that receives a predicate and a pair of values *a* and *b*. If the predicate is true, then a is returned. Otherwise, b is returned. The predicate usually is generated with a previous comparison or logical operations and is represented as a binary secret sharing. The secret *select* operation works as a computation listed in Equation 1, where the predicate is transformed into 0 (false) and 1 (true) as a multiplier that demands converting binary secret sharing to arithmetic secret sharing. The conversion requires communication and is expensive [15]. We observe that a predicate may be used by multiple *select* operations when training some convolutional neural networks. Once SPU frontend detects this pattern, a *prefer_a* operation that explicitly converts binary secret sharing to arithmetic secret sharing will be inserted before the first select operation to reduce redundant conversions (as shown in Figure 7).

$$Select(pred, a, b) = b + pred * (a - b) \tag{1}$$

Max-pooling transformation. Max-pooling is a widely-used layer in convolutional neural networks, usually connected behind the convolutional layer for downsampling input features. In the forward propagation stage, max-pooling needs to find the maximum value in a window of values. In the backpropagation stage, max-pooling needs to replace the maximum value with its gradient while other values are set to zeros. We observe that the two stages are computed by two independent operations (i.e., *reduce_windows* and *select_and_scatter* in Figure 8) when ML frameworks train models. Although the *reduce_window* operation has found the maximum value in a window, *select_and_scatter* will do the same to find the index of the maximum value. Redundant and expensive comparisons would be called in both operations. Therefore, SPU frontend transforms this computation pattern to two new operations we proposed in PPHLO, i.e., *argmax* and *maxpool_scatter*. In the *argmax* operation, we will get the maximum value and its index in the window. The index can be directly reused by *maxpool_scatter* operation. We use a one-hot vector to represent the index. The maximum value can be updated by multiplying the index by the gradient.

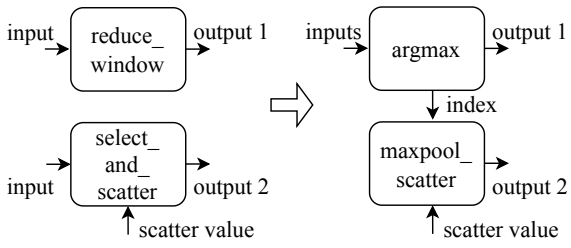


Figure 8: Max-pooling transformation.

In this section, we share observations on generated PPHLO from the ML computation graph and operation optimization strategies we implemented. Optimization techniques for ML computational graphs are much more mature with the efforts of countless experts and engineers. However, MPC computations introduce a different cost model compared to CPU or GPU computing, which enables us to adopt novel optimization techniques on PPHLO. We believe that more optimization opportunities are waiting to be discovered in this new interdisciplinary field.

3.6 Backend

SPU backend consists of multiple computing nodes, and each node contains a runtime to execute PPHLO operations. The number of nodes should match the parties of the underlying MPC protocol. The design goal of SPU backend is scalable to support different MPC protocols with a pluggable experience and efficiently execute PPHLO operations as MPC protocols. This section describes the operation dispatching mechanism and runtime optimizations used in SPU backend.

3.6.1 Operation Dispatching

We design the layered dispatching mechanism to achieve extensibility to different MPC protocols, as shown in Figure 9. For each PPHLO operation, the SPU runtime will first dispatch it to the PPHLO layer, which has a one-to-one mapping function acting as the operation entrance. The PPHLO mapping function will further dispatch the operation to fine-grained HAL (Hardware Abstraction Layer) functions. We borrow the concept of HAL from traditional operating system implementation which is mean to eliminate the boundary between hardware and software. We use HAL to hide the MPC implementation details from PPHLO operations. Specifically, at the HAL level, the SPU runtime will decompose an operation into a set of MPC-primitive functions according to the data type and visibility of operands. As the example in Figure 9, a secret fixed-point number multiplication will be decomposed to a secret integer multiplication function and a truncation function. Each MPC-primitive function will be finally dispatched to the MPC layer, corresponding to a specific implementation of fundamental MPC protocols. Adding a new MPC protocol in SPU only needs to implement the MPC-primitive function set. When users configure a new backend protocol, SPU's runtime will dispatch PPHLO operations to the new MPC implementation.

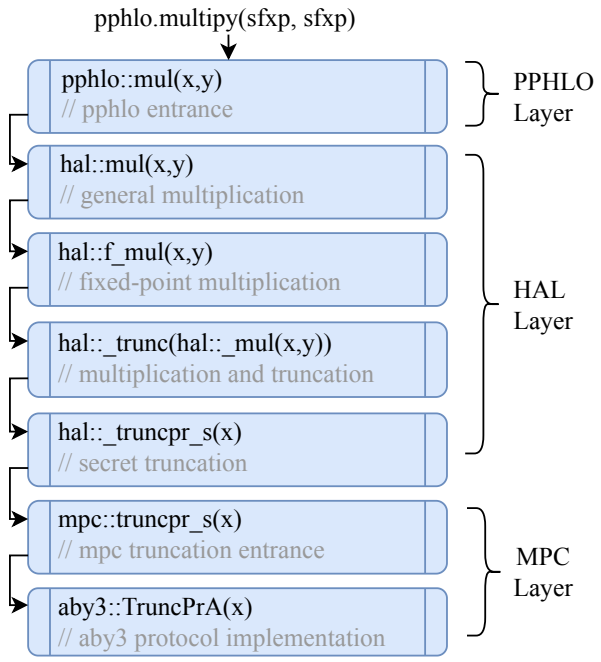


Figure 9: The dispatching path from a PPHLO operation to an MPC protocol in SPU. Different protocols can reuse the same PPHLO/HAL layer code and diverge at the final MPC layer.

3.6.2 Runtime Optimizations

We employ the following techniques in SPU runtime to achieve high performance.

Vectorization. Vectorization is a standard technique used on CPUs supported by SIMD (Single Instruction, Multiple Data) instructions. Applying one instruction to multiple data enables parallelism and improves program execution efficiency. SPU implements a similar vectorization mechanism by running one operation on a list of data to reduce the number of executed operations. For example, there are two operations *mul(a,b)* and *mul(c,d)* where *mul* stands for an element-wise multiplication operation and a, b, c, d are tensors. SPU will pack a, b, and c, d together and execute one *mul* operation on these tensors. As MPC multiplication needs communications, SPU can reduce the number of communication rounds through vectorization.

Streaming. Many MPC operations involve both intensive network I/O activities and local computations. If such an MPC operation processes a very large tensor, a more efficient method is to tile the tensor into sub-tensors and use multiple operations to process them concurrently. We illustrate this problem with the toy model in Figure 10. An ML model training stage consists of many iterations. When a processed tensor is enormous, the processing operations repeatedly block network I/O and local computing, affecting the overall execution efficiency. Suppose we tile the tensor into two small tensors and execute them concurrently. In that case, multiple sub-tensors and sub-operations can significantly improve network

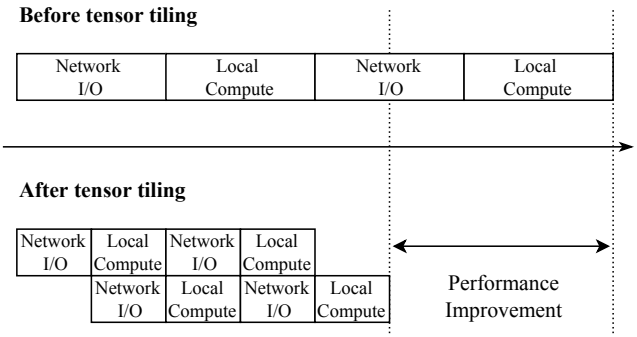


Figure 10: Streaming for MPC operations.

and computing resource utilization, thereby shortening execution time.

Concurrency. SPU supports both intra- and inter-operation concurrency to gain performance benefits. For intra-operation concurrency, most PPHLO operations consist of more than one HAL function. We implement these operations with well-established engineering experience and execute multiple functions without data dependencies in different threads. Taking the implementation of *rsqrt* operation as an example, we follow the protocol proposed by Lu et al. [41] to compute the inverse square root of a tensor x. The calculation consists of a fractional part computation and an exponential part computation which can be computed independently in two threads.

For inter-operation concurrency, SPU uses an aggressive strategy at the PPHLO graph granularity to prevent operations with communications blocking the use of computing resources. When executing PPHLO computation graph, SPU runtime launches as many operations as possible asynchronously. Once an operation's dependencies are completed, this operation will be scheduled for execution.

4 Implementation and Evaluation

We implement SPU frontend compiler based on MLIR (Multi-Level Intermediate Representation) [36], a compiler infrastructure for domain-specific computations. PPHLO is implemented as a new MLIR dialect for MPC computations. Frontend optimizations are implemented as compiler passes. We develop SPU backend with modern standard C++, and its computing nodes communicate through a high-performance RPC library, bRPC [1]. The SPU C++ library is binding to Python interfaces exposed to application developers as a Python module (Section 3.3). Currently, we provide users with three built-in MPC protocols, i.e., the semi-honest implementations of a three-party protocol ABY3 [43] and a N-party protocol SPDZ2k [13], and a two-party protocol Cheetah [24]. Our overall code base contains more than 50k LOC of C++ and 3k LOC of Python.

In the rest of this section, we evaluate SPU on performance and user-friendliness. Evaluations are done on three Alibaba Cloud *ecs.g7.xlarge* instances with 4 vCPU and 16GB RAM each. We complete evaluations under local area network (LAN, 10.1Gbps bandwidth and 0.1ms round-trip time) and wide area network (WAN, 300Mbps bandwidth and 40ms round-trip time) settings.

4.1 Performance

To evaluate SPU's performance, we compare SPU to general MPC-enabled PPML frameworks rather than some specific protocol implementations. We use SPU and three optimized frameworks (i.e., MP-SPDZ [29], TF Encrypted [14], and CrypTen [33]) to train four neural network models for image classification on the MNIST dataset. The models are trained privately with the encrypted training dataset and revealed to evaluate on the plaintext validation dataset. All frameworks train models with three ML optimizers, i.e., SGD, Adam [31], and AMSGrad [48] (except CrypTen which does not support Adam and AMSGrad). Using different optimizers to train models results in distinct computation graphs, causing varying computation costs that are noticeably divergent between MPC scenarios and plaintext training. Consequently, evaluating diverse optimizers showcases SPU's performance extensibility.

The selected four models have been widely used in related literature for evaluations [30, 40, 44, 49, 56, 57], and their detailed architectures are listed in Appendix A.1 (Table 2, 3, 4, and 5). We follow the numbering (from A to D) given by Wagh et al. [56] to refer to these models. All training experiments use a semi-honest three-party MPC (3PC) protocol, which is widely used and supported by all frameworks.

Table 1 reports the classification accuracy and seconds per batch when training 5 epochs with a batch size of 128. MP-SPDZ data is collected by running scripts provided by its author Keller [30] (commit 0f7020d). TF Encrypted data is collected by directly running scripts provided in its code repository (commit 51de98f). CrypTen data is collected by running an adaption of its official example *mpc_autograd_cnn* (commit 909df45). For SPU, we write JAX programs to train models and run these programs on SPU to collect data.

In Table 1, CrypTen has a significant gap with the other three frameworks in terms of both training speed and classification accuracy. One possible reason for this may be that CrypTen is primarily implemented in Python. Besides, CrypTen differs from the other three works in that it does not strictly employ a standard semi-honest 3PC protocol based on replicated secret sharing [6]. The protocol CrypTen uses can support any number of participants ($N \geq 2$), and we evaluate its performance in the three-party scenario.

Therefore, in order to ensure fairness, our comparisons in this section will mainly focus on SPU, MP-SPDZ, and TF Encrypted. For accuracy, the three frameworks all get high and close results. These is no single framework can achieve optimal results in every configurations. For training time, MP-SPDZ has better results under LAN while losing its advantages under WAN compared to TF Encrypted . The possible reason is that MP-SPDZ implements a more efficient multi-threaded kernel for operation execution, so it benefits from the intensive local computations under LAN. However, when network I/O communications become the bottleneck under WAN, TF Encrypted which relies on the underlying TensorFlow for graph scheduling works better.

Compared with MP-SPDZ and TF Encrypted, SPU achieves the fastest training on 11 out of 12 configurations under LAN and all configurations under WAN. Under LAN, SPU's advantage over MP-SPDZ is minor but achieves 1.4-4.6× faster training than TF Encrypted. Under WAN, SPU achieves up to 4.1× faster training than MP-SPDZ and up to

Table 1: The accuracy and seconds per batch of training four neural network models on the MNIST dataset with SGD/Adam/AMSGrad optimizer in four MPC-enabled PPML frameworks. M, T, C, and S are abbreviations of MP-SPDZ [29], TF Encrypted [14], CrypTen [33], and our SPU, respectively. CrypTen does not support Adam and AMSGrad as of the time we write this paper.

Network	Accuracy				Seconds per Batch (LAN)				Seconds per Batch (WAN)			
	M	T	C	S	M	T	C	S	M	T	C	S
A (SGD)	96.8%	96.4%	92.7%	**96.9%**	0.16	0.19	1.43	**0.12**	8.94	**4.60**	58.68	**4.60**
A (Adam)	**97.5%**	97.2%	N/A	97.4%	0.42	0.56	N/A	**0.39**	17.72	12.60	N/A	**7.67**
A (AMSGrad)	**97.6%**	97.4%	N/A	97.5%	0.42	0.71	N/A	**0.41**	18.28	13.26	N/A	**7.68**
B (SGD)	98.1%	98.3%	96.5%	**98.4%**	**1.00**	4.82	25.62	1.04	34.70	15.66	230.15	**9.87**
B (Adam)	97.9%	**98.7%**	N/A	**98.7%**	1.13	4.90	N/A	**1.12**	44.92	18.18	N/A	**11.15**
B (AMSGrad)	98.7%	**98.8%**	N/A	98.6%	1.13	4.78	N/A	**1.12**	45.73	18.08	N/A	**11.23**
C (SGD)	98.5%	**98.9%**	97.3%	98.8%	2.10	7.23	34.06	**1.81**	50.05	22.41	272.11	**12.98**
C (Adam)	98.8%	**99.0%**	N/A	98.9%	2.92	8.33	N/A	**2.37**	67.03	49.51	N/A	**22.87**
C (AMSGrad)	**99.2%**	98.9%	N/A	99.1%	2.94	8.93	N/A	**2.37**	67.49	51.06	N/A	**22.53**
D (SGD)	97.0%	**97.6%**	95.7%	97.2%	0.23	0.39	1.77	**0.22**	11.20	5.35	59.44	**4.89**
D (Adam)	97.8%	**98.0%**	N/A	97.7%	0.45	0.69	N/A	**0.43**	19.87	12.12	N/A	**7.66**
D (AMSGrad)	**98.3%**	97.5%	N/A	97.9%	0.45	0.81	N/A	**0.43**	20.42	12.76	N/A	**7.66**

2.3× faster training than TF Encrypted. Overall, the evaluation results demonstrate that SPU achieves state-of-the-art performance by combining the two aspects of WAN and LAN.

We further analyze the performance benefits of SPU. Although we implement a series of compiler passes to optimize the computation graph, it should be noted that these optimizations are workload-dependent, and not all optimizations will be effective for a specific workload. Taking training Network C with the Adam optimizer, which has the most complex computation graph, as an example. We find that when all compiler passes are disabled, the training time is 2.0× slower under LAN (4.63 versus 2.37 seconds) and is 1.9× slower under WAN (43.49 versus 22.87 seconds). Additionally, we find that nearly all performance benefits come from two frontend optimizations, i.e., max-pooling transformation and inverse square root transformation. This phenomenon does not mean that other implemented optimizations are meaningless, as we observe that those compiler passes are more effective on other workloads with corresponding computational patterns, such as training decision tree models.

Another conclusion we can draw is that the backend runtime also plays a significant role in contributing to SPU's performance improvement. Network A does not have a max-pooling layer and the SGD optimizer also does not involve the *rsqrt* operation. As a result, the two optimizations mentioned above do not apply to training Network A with SGD. However, SPU also achieves state-of-the-art in this experimental setting. Therefore, we believe SPU's high-performance benefits from collaborative frontend/backend implementations.

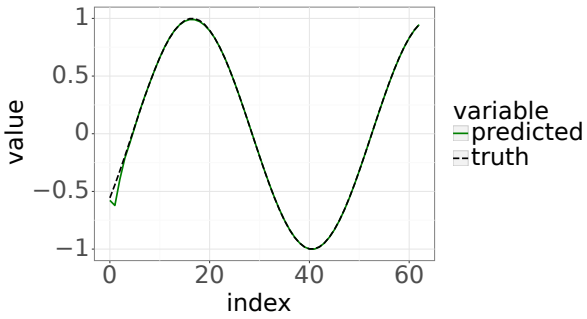
4.2 User-friendliness

This section evaluates SPU's user-friendliness through its compatibility with ML applications from different mainstream ML frameworks. We select two ML training programs from well-known open-source JAX projects and run them on SPU to train models privately. We found that only minor modifications to these programs are required to run them on SPU with acceptable overhead and achieve results comparable to plaintext training on CPUs. Besides, we test SPU's feasibility to run TensorFlow and PyTorch programs. These experiments show that SPU can be easily extended to other ML models and frameworks. The evaluations for SPU also use the same setting (a semi-honest 3PC protocol) as in Section 4.1 under LAN. The evaluations for plaintext training and prediction on CPUs run on a single cloud server.

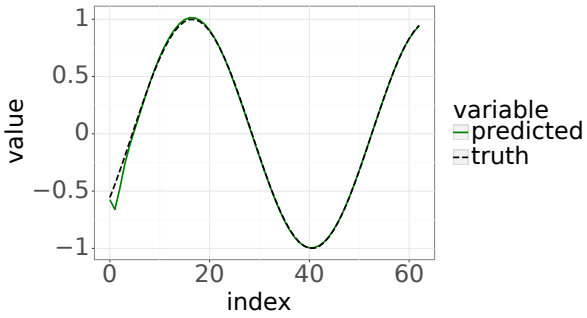
4.2.1 Long Short-Term Memory

This example of training a Long Short-Term Memory (LSTM) model [23] comes from Haiku [21], a JAX neural network library developed by DeepMind. LSTM is a recurrent neural network model for processing sequential data such as text or speech. This example trains an LSTM model to predict time series, using the data generated from a sine wave for training and validation. The model is trained privately with the encrypted training dataset and revealed to evaluate on the plaintext validation dataset. We modify about 8 lines of the example's source code to enable SPU to run the training program.

The vanilla JAX program takes 3.01 seconds to train 2001



(a) JAX.



(b) SPU.

Figure 11: Predictions with LSTM models trained by JAX and SPU.

steps, reducing the training loss from 0.45737 to 0.00067. SPU takes 7177.83 seconds (2384.7× slowdown) to reduce the training loss to 0.00099. Figure 11 shows that we use JAX and SPU-trained models to predict validation data with ground truth. The SPU-trained model achieves similar prediction results with high accuracy compared to the JAX model that is trained in plaintext.

4.2.2 Variational Auto-Encoder

This section describes a Variational Auto-Encoder [32] (VAE) model training example from Flax [20], a JAX neural network library developed by Google. VAE is a generative model which can map high-dimensional input space into low-dimensional latent space and regenerate the input from the latent representation. This example trains a VAE model to compress and regenerate images from the MNIST dataset. The model is trained privately with the encrypted training images and revealed to evaluate on the plaintext testing images. We modify about 22 lines of code to enable SPU to run the training program.

It takes JAX and SPU 214 and 9131 seconds (42.7× slowdown) to train 5 epochs with a batch size of 128,[2] reducing the training loss from 535 to 106. Figure 12 shows that using the model trained by SPU to reconstruct MNIST digits has a comparable result to JAX.

[2]The times include an extra evaluation on testing dataset in each iteration.

(a) JAX.

(b) SPU.

Figure 12: Reconstruct MNIST digits with VAE models trained by JAX and SPU. The digits above are original inputs.

4.2.3 Beyond JAX

Technically, SPU is able to support any ML frameworks that can be translated to HLO. In this section, we validate SPU's feasibility to support TensorFlow and PyTorch as frontend ML frameworks. For TensorFlow programs, SPU Python APIs can also be used directly on ML training or prediction functions made of TensorFlow functions. SPU will call *tf.function* API provided by TensorFlow to compile these composite functions into HLO as SPU frontend inputs. For PyTorch programs, SPU relies on the Torch-MLIR [2] project to convert them into HLO, which SPU can further consume.

We train a TensorFlow logistic regression model with the diagnostic Wisconsin breast cancer dataset [52]. The model is trained privately on SPU with the encrypted training dataset and revealed to evaluate on the plaintext testing dataset. SPU achieves the same ROC-AUC (Area Under the Receiver Operating Characteristic Curve) [16] of 0.99 as the plaintext training result on CPU. Training times on CPU and SPU are 0.067 and 1.121 seconds (16.7× slowdown).

As for PyTorch, we use the same dataset to train a linear classification model on the plaintext data and run predictions with jointly encrypted features on SPU (the model weights are not protected in this example). Experimental results show that compared to plaintext prediction on vanilla PyTorch, SPU achieves the same ROC-AUC of 0.97 with a 345.7× speed slowdown (0.05186 versus 0.00015 seconds). Overall, these results demonstrate SPU is feasible to support different ML frameworks.

5 Limitations and Discussion

This section discusses some known issues of SPU. SPU uses the fixed-point representation to encode decimal numbers like other MPC-based ML systems, which leads to two limitations. First, fixed-point numbers have limited precision and range compared to floating-point numbers. This problem will cause running some ML programs on SPU to get incorrect results. We can mitigate this problem by using more fractional bits to

encode fixed-point numbers in SPU. Second, some function implementations rely on floating-point number representations (such as JAX *random.normal*), which will cause SPU to compute these functions with unexpected results. We will try to provide a library overwriting these functions in the future.

Besides, SPU currently does not support secret conditions as some operations (such as *while*) may use. In most cases, these tensors are not data that need to be protected. Developers can implement these values as public tensors.

6 Related Work

In recent years, there has been a line of works studying applying MPC techniques for PPML [9,24,27,34,43,44,49,56,57]. Most of these works focus on protocol optimizations and innovations, improving MPC-enabled PPML's performance and making it a practical solution. These works diverge on security models (malicious or semi-honest adversaries, honest-majority or dishonest-majority), secret sharing schemas, the number of supported parties, and implementation details of basic operations. These protocol innovations are orthogonal to SPU, which can implement them as the underlying protocols.

Another bunch of works try to reduce MPC usage difficulty for non-MPC experts by implementing general-purpose MPC compilers. These compilers convert functions written in high-level or domain-specific languages to MPC circuits, which are later executed by backend runtime in an MPC manner. Hastings et al. [18] have a detailed survey on these compilers. However, these works are not tailored-made for ML scenarios. Using them to develop complex and efficient ML programs takes significant work.

The most relevant works to SPU are TF Encrypted [14], CrypTen [33], and MP-SPDZ [29]. TF Encrypted and CrypTen provide programming interfaces similar to TensorFlow and PyTorch in their Python modules. Refactoring an ML program into the PPML version must replace original ML APIs with TF Encrypted/CrypTen APIs corresponding to MPC implementations. The frontend of MP-SPDZ is Python. Users write ML programs based on Python APIs provided by MP-SPDZ, which compiles programs to byte code and runs in an MPC manner. Compared with these frameworks, SPU runs programs written in existing ML frameworks. Besides, SPU can support more than one ML framework.

There are other PPML frameworks developed based on Trusted Execution Environments (TEE) [47] or federated learning (FL) [28]. TEE-based solutions require special hardware and are vulnerable to side-channel attacks [10,54]. FL also enables multiple participants to jointly and privately train a model. In the classic FL scenario, each participant performs local gradient computations on the plaintext datasets, and then a centralized server aggregates the model parameters from participants. As the original input data remains within the owner's domain throughout the training process, FL can be considered as a PPML solution. However, some works have already shown that even only exchanging model parameters may also threaten the original input data [35,42,61].

Compared with FL frameworks, SPU provides end-to-end privacy protection based on provable MPC techniques. MPC does not necessarily require an independent server responsible for model aggregation. Participants' data is first encrypted and then fed into SPU. SPU then performs computations on the encrypted data (such as gradient updates) and trains a model, which is also kept in encryption. Finally, the model is reconstructed to plaintext and revealed to some designated parties. In addition to model training, another use case for SPU is private model inference, in which one party protects the input data, and the other protects the model parameters.

More advanced FL frameworks have been proposed in recent years to improve FL's security. Chen et al. [12] introduce MPC techniques into model aggregation to resist generative adversarial network attacks [22]. HybridAlpha [58] uses functional encryption [7] to prevent curious aggregators and colluding participants from inferring private data. Both approaches require a model aggregator and an additional trusted third party, which are not necessarily required in SPU. Besides, Chen et al. [12] mainly target convolutional neural networks, while SPU is not limited to specific model types. Triastcyn et al. [53] propose FedGP to replace participants' original data with artificial data by training generative adversarial networks [17]. However, their approach is limited to protecting image data and lacks a theoretical security guarantee. Compared with FedGP, SPU has no restrictions on the protected data types, and its security guarantee is based on provable MPC techniques.

7 Conclusion

In this paper, we propose SPU, a compiler and runtime suite, which converts ML programs into an MPC-specific IR and executes the IR in an MPC manner. Using the Python APIs provided by SPU, users can achieve privacy-preserving ML training and prediction by writing programs in mainstream ML frameworks. We believe that using SPU can significantly lower the threshold for users to achieve privacy protection and promote the development of the entire PPML community.

Acknowledgments

We would like to thank our shepherd, Sara Bouchenak, and the anonymous reviewers for their invaluable comments. We would also like to acknowledge the insightful feedback provided by Yu Luo on the early version of our paper. We extend our appreciation to Wen-jie Lu, Zhicong Huang, and Cheng Hong for their significant contributions to SPU. Lastly, we thank all members of the SecretFlow team for their support throughout this project.

References

[1] better rpc. https://github.com/apache/incubator-brpc, 2023.

[2] The torch-mlir project. https://github.com/llvm/torch-mlir, 2023.

[3] Xla operation semantics. https://www.tensorflow.org/xla/operation_semantics, 2023.

[4] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla, 2023.

[5] Martin 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, 2016.

[6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 805–817, 2016.

[7] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *Theory of Cryptography*, pages 253–273, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] 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.

[9] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511, 2019.

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. *IEEE Secur. Priv.*, 18(3):28–37, 2020.

[11] 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.

[12] Zhenzhu Chen, Anmin Fu, Yinghui Zhang, Zhe Liu, Fanjian Zeng, and Robert H. Deng. Secure collaborative deep learning against gan attacks in the internet of things. *IEEE Internet of Things Journal*, 8(7):5839–5849, 2021.

[13] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spdz2k: Efficient mpc mod 2k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 769–798, Cham, 2018. Springer International Publishing.

[14] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. Private machine learning in tensorflow using secure computation. *arXiv preprint arXiv:1810.08130*, 2018.

[15] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[16] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Commun. ACM*, 63(11):139–144, oct 2020.

[18] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: general-purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[20] Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2020.

[21] Tom Hennigan, Trevor Cai, Tamara Norman, and Igor Babuschkin. Haiku: Sonnet for JAX, 2020.

[22] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: Information leakage from collaborative deep learning. In *Proceedings of*

the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, page 603–618, New York, NY, USA, 2017. Association for Computing Machinery.

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

[24] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure Two-Party deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[26] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

[27] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, Baltimore, MD, August 2018. USENIX Association.

[28] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista A. Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. *Found. Trends Mach. Learn.*, 14(1-2):1–210, 2021.

[29] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[30] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In *International Conference on Machine Learning*, pages 10912–10938. PMLR, 2022.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[32] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[33] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubhabrata Sengupta, Mark Ibrahim, and Laurens van der Maaten. CrypTen: Secure multi-party computation meets machine learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[34] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *2020 IEEE Sym-*

posium on Security and Privacy (SP), pages 336–353, 2020.

[35] Maximilian Lam, Gu-Yeon Wei, David Brooks, Vijay Janapa Reddi, and Michael Mitzenmacher. Gradient disaggregation: Breaking privacy in federated learning by reconstructing the user participant matrix. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 5959–5968. PMLR, 2021.

[36] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr 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.

[37] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[38] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44, 2019.

[39] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, 2021.

[40] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 619–631, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Wen-jie Lu, Yixuan Fang, Zhicong Huang, Cheng Hong, Chaochao Chen, Hunter Qu, Yajin Zhou, and Kui Ren. Faster secure multiparty computation of adaptive gradient descent. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, PPMLP'20, page 47–49, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 691–706. IEEE, 2019.

[43] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 35–52, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017.

[45] 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 Yang, Zach 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*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[46] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol secure Two-Party computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2165–2182. USENIX Association, August 2021.

[47] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. Securetf: A secure tensorflow framework. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, pages 44–59, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[49] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 707–721, New York, NY, USA, 2018. Association for Computing Machinery.

[50] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[51] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[52] Nick Street, William Wolberg, and O Mangasarian. Nuclear feature extraction for breast tumor diagnosis. *Proc. Soc. Photo-Opt. Inst. Eng.*, 1993, 01 1999.

[53] Aleksei Triastcyn and Boi Faltings. Federated generative privacy. *IEEE Intelligent Systems*, 35(4):50–57, 2020.

[54] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 991–1008, USA, 2018. USENIX Association.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[56] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies*, 2019.

[57] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. 2021.

[58] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, AISec'19, page 13–23, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.

[60] Tjalling J. Ypma. Historical development of the newton–raphson method. *SIAM Review*, 37(4):531–551, 1995.

[61] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14747–14756, 2019.

A Supplementary Materials

A.1 Evaluated Neural Network Models

This section describes models used in Section 4.1. Network A is defined in [44]. Network B is defined in [40]. Network C is defined in [37]. Network D is defined in [49].

Table 2: Architecture of Network A (SecureML [44] model).

Layer	Input	Description	Output
Fully Connected	784	784×128 matrix multiplication	128
ReLU	128	Element-wise ReLU on input	128
Fully Connected	128	128×128 matrix multiplication	128
ReLU	128	Element-wise ReLU on input	128
Fully Connected	128	128×10 matrix multiplication	10

Table 3: Architecture of Network B (MiniONN [40] model).

Layer	Input	Description	Output
Convolution	$1 \times 28 \times 28$	5×5 kernel, 1×1 stride	$16 \times 24 \times 24$
MaxPooling	$16 \times 24 \times 24$	2×2 kernel	$16 \times 12 \times 12$
ReLU	$16 \times 12 \times 12$	Element-wise ReLU on input	$16 \times 12 \times 12$
Convolution	$16 \times 12 \times 12$	5×5 kernel, 1×1 stride	$16 \times 8 \times 8$
MaxPooling	$16 \times 8 \times 8$	2×2 kernel	$16 \times 4 \times 4$
ReLU	$16 \times 4 \times 4$	Element-wise ReLU on input	$16 \times 4 \times 4$
Fully Connected	256	256×100 matrix multiplication	100
ReLU	100	Element-wise ReLU on input	100
Fully Connected	100	100×10 matrix multiplication	10

Table 4: Architecture of Network C (LeNet [37] model).

Layer	Input	Description	Output
Convolution	$1 \times 28 \times 28$	5×5 kernel, 1×1 stride	$20 \times 24 \times 24$
MaxPooling	$20 \times 24 \times 24$	2×2 kernel	$20 \times 12 \times 12$
ReLU	$20 \times 12 \times 12$	Element-wise ReLU on input	$20 \times 12 \times 12$
Convolution	$20 \times 12 \times 12$	5×5 kernel, 1×1 stride	$50 \times 8 \times 8$
MaxPooling	$50 \times 8 \times 8$	2×2 kernel	$50 \times 4 \times 4$
ReLU	$50 \times 4 \times 4$	Element-wise ReLU on input	$50 \times 4 \times 4$
Fully Connected	800	800×500 matrix multiplication	500
ReLU	500	Element-wise ReLU on input	500
Fully Connected	500	500×10 matrix multiplication	10

Table 5: Architecture of Network D (Chameleon [49] model).

Layer	Input	Description	Output
Convolution	$1 \times 28 \times 28$	5×5 kernel, 2×2 stride	$5 \times 14 \times 14$
ReLU	$5 \times 14 \times 14$	Element-wise ReLU on input	$5 \times 14 \times 14$
Fully Connected	980	980×100 matrix multiplication	100
ReLU	100	Element-wise ReLU on input	100
Fully Connected	100	100×10 matrix multiplication	10

B Artifact Appendix

B.1 Abstract

SecretFlow-SPU is an open-source framework designed for privacy-preserving machine learning. This artifact contains

the source code of SecretFlow-SPU, along with documentation for reproducing the experiments reported in this paper. Additionally, we provide scripts and a Docker container image to quickly build the experimental settings.

B.2 Scope

The artifact includes experiments for secure neural network training, secure Variational Auto-Encoder (VAE) training, and secure Long Short-Term Memory (LSTM) training using JAX. We also provide two simple TensorFlow and PyTorch demos. These experiments cover all we reported results in the paper.

B.3 Contents

README.md describes the artifact and provides a road map for evaluation. For more details on the SecretFlow-SPU repo's directory layout, please refer to *REPO_LAYOUT.md* under the base directory.

B.4 Hosting

The artifact is available at https://github.com/secretflow/spu (branch *atc23_ae*).

B.5 Requirements

SecretFlow-SPU has no special hardware requirements. To reproduce our results, users should have at least three servers that are connected within a high-performance network. We have done our evaluations on three Alibaba Cloud ecs.g7.xlarge cloud servers with 4 vCPU and 16GB RAM each. The CPU model is Intel(R) Xeon(R) Platinum 8369B CPU @ 2.70GHz. We evaluated SecretFlow-SPU on Ubuntu 20.04.5 LTS with Linux kernel 5.4.0-125-generic. Technically, SecretFlow-SPU is supported to run on any Linux servers with software requirements described in *CONTRIBUTING.md*.

Portunus: Re-imagining Access Control in Distributed Systems

Watson Ladd[*,†] *Akamai*
Tanya Verma[*] *Cloudflare*
Marloes Venema *University of Wuppertal*
Armando Faz-Hernández *Cloudflare*
Brendan McMillion[†]
Avani Wildani *Cloudflare*
Nick Sullivan *Cloudflare*

Abstract

TLS termination, which is essential to network and security infrastructure providers, is an extremely latency-sensitive operation that benefits from access to sensitive key material close to the edge. However, increasing regulatory concerns prompt customers to demand sophisticated controls on where their keys may be accessed. While traditional access-control solutions rely on a highly-available centralized process to enforce access, the round-trip latency and decreased fault tolerance make this approach unappealing. Furthermore, the desired level of customer control is at odds with the homogeneity of the distribution process for each key.

To solve this dilemma, we have designed and implemented Portunus, a cryptographic storage and access control system built using a variant of public-key cryptography called attribute-based encryption (ABE). Using Portunus, TLS keys are protected using ABE under a policy chosen by the customer. Each server is issued unique ABE keys based on its attributes, allowing it to decrypt only the TLS keys for which it satisfies the policy. Thus, the encrypted keys can be stored at the edge, with access control enforced passively through ABE. If a server receives a TLS connection but is not authorized to decrypt the necessary TLS key, the request is forwarded directly to the nearest authorized server, further avoiding the need for a centralized coordinator. In comparison, a trivial instantiation of this system using standard public-key cryptography might wrap each TLS key with the key of every authorized data center. This strategy, however, multiplies the storage overhead by the number of data centers. Deployed across Cloudflare's 400+ global data centers, Portunus handles millions of requests per second globally, making it one of the largest deployments of ABE.

1 Introduction

Transport Layer Security (TLS) is a cryptographic protocol widely used to secure communication and protect data integrity between clients, such as browsers, and servers, who host the websites. In a TLS handshake, the server presents a certificate—containing its public key—to the client, and uses the associated private signing key to create a digital signature. This verifies the website's authenticity and creates a secure connection.

Seeking enhanced performance and security, website operators often enlist the services of infrastructure providers like Content Delivery Networks (CDNs). These providers—offering services such as DDoS protection, load balancing, and caching—run on globally distributed data centers to ensure low latency and high performance, and to maintain availability. They also need to be able to inspect the TLS connection between clients, who are the end users of their customer's websites, and their customer's servers. This process of intercepting a TLS connection at an intermediary point in the network is called TLS termination. To handle TLS termination on behalf of their customers, service providers require access to the private signing key for their respective websites.

However, customers utilizing these services have different degrees of comfort concerning the use of their key material across data centers. For example, European customers may stipulate key storage exclusively within the European Union. Another might demand key storage only in data centers secured with bulletproof glass and laser alarm systems. These customers would like providers to control access to their key material based on geographical and security properties. Given that the TLS handshake is in the critical path of establishing a connection to a website, any latency introduced by key access control methods could significantly disrupt service quality. Additionally, for larger infrastructure providers handling millions of TLS terminations per second, minimizing computational overhead from the access control method is essential to scalability.

Unfortunately, traditional access control mechanisms fall short in this endeavor. Centralized methods of access control [41] require edge data centers to communicate with the network's control plane to access specific keys, leading to an expensive round-trip which adds latency and reduces reliability. Alternately, access control using standard public-key encryption provides low latency by assigning

[*] Equal contribution
[†] Work done while at Cloudflare

unique encryption keys to each data center and encrypting the customer's private signing key with the keys of each data center that complies with the access policy. This encrypted data can then be disseminated across all edge data centers in advance of connection requests, reducing latency. However, this strategy becomes rather complex to manage in the face of heterogeneous policies and large scale. The ciphertext size grows in proportion to the number of data centers, creating large overheads. Newly added centers cannot participate in establishing TLS connections unless the customer's signing key is re-encrypted with their newly issued encryption keys.

To address these issues, we required a more direct way to enforce access control through cryptography. Our first attempt [52, 53] combined identity-based encryption [11, 51] and broadcast encryption [25], but ultimately was too inflexible and limited in the types of access policies it could support. Spurred by these restrictions, we created Portunus. Portunus uses a variant of traditional public key cryptography called ciphertext-policy attribute-based encryption (CP-ABE) [9], which can implement fine-grained access control on a cryptographic level. CP-ABE is a a variant of the more general notion of attribute-based encryption (ABE), which was first proposed by Sahai and Waters [46] as a type of public-key encryption in which the keys and ciphertexts are associated with attributes instead of individual users. Concretely, CP-ABE links the secret keys to the attribute set of the key holders, and the ciphertexts to access policies that govern which key holders can decrypt them. Those policies are determined by the encryptor, who can therefore manage access to their data in the spirit of attribute-based access control (ABAC) [38].

We have adopted Portunus at scale. Using Portunus, TLS keys are encrypted using an X25519 key that serves as a data encryption key, which we call the *policy key*. This policy key is further encrypted using ABE under a policy chosen by the customer. Both the encrypted customer keys and policy keys are stored in a globally replicated database present on every machine at Cloudflare. Each edge machine has attributes determined by a database mapping its core cryptographic identity to a set of attributes, e.g., country and region. Edge machines are issued unique ABE secret keys by a key generation authority run in the control plane, allowing them to decrypt only the policy keys that they are authorized to access based on their attributes. Thus, both the encrypted customer keys and policy keys can be stored at the edge, with access control enforced passively through ABE. If a server receives a TLS connection but is not authorized to decrypt the necessary TLS key, the request is forwarded directly to the nearest authorized server, further avoiding the need for a centralized coordinator. As new machines are added, they automatically have access to the keys to which they are permitted by the policy.

While decryption in ABE is more computationally expensive than its equivalent in traditional public key cryptography, we are able to significantly mitigate its impact through session resumption and caching decrypted policy keys.

Adopting CP-ABE as a storage-layer access control solution means that all nodes share the same data, simplifying the distribution process. It also makes it easy for newly added nodes to take up the burden of satisfying requests. Furthermore there are no centralized components whose failure would lead to breaks in the availability of the system. Cryptographically enforced access control is inherently less coupled and more fault tolerant than a centralized system would be.

Our core contributions are:

1. Portunus, a real-world deployment of an ABE-based access control system for key management. Although several works have shown interest in using ABE [22, 23, 32, 48], few have resulted in large-scale real-world deployments.

2. A discussion of the practical costs and benefits of such a scheme, concluding that it is effective in solving distributed access control

3. Lessons learned for future use of CP-ABE by engineers and for ABE researchers about real-world requirements

2 Requirements

In the design process for Portunus, we identified a series of requirements arising from customer needs, internal engineering demands, and the experience of operating the predecessor of Portunus, Geo Key Manager [52, 53].

Low computational overhead: As TLS handshakes can happen at extremely high volumes for legitimate reasons, it is essential that we not add significant computational overhead to the responding process.

Rotation capable: It should be easy to rotate encryption keys used in the system. Key rotation is the practice of systematically replacing cryptographic keys with new ones periodically, to limit the amount of data exposed by the compromise of a particular key. A rotation ensures that newly-uploaded TLS signing keys are not decryptable by machines that have not been updated with new key material.

Recovery from strong attackers: We assume an attacker that is capable of compromising multiple edge machines and reading the database of certificates and associated signing keys. We would like this attacker to be unable to continue impersonating sites after their access is removed, unless the site's certificate was decryptable on the machines they compromised. We also want subsequent certificates not to be decryptable by the attacker after key rotation.

Flexible attributes: Historically, the set of attributes customers want changes over time, such as when a new compliance standard is introduced. Accommodating these changes in the prior system, Geo Key Manager, took considerable work.

Flexible policies: From experience, we know that customers and internal services would need a wide range of policies. Even

if the eventual product did not expose the full expressiveness, future developments would be difficult to anticipate.

Limited storage: Quicksilver, Cloudflare's configuration management system [43], has limited space because it duplicates all data across all machines globally. To preserve fault tolerance and the ability to serve requests quickly from all machines, our system needs to minimize storage overhead.

Uniformity of data: Quicksilver employs a homogeneous tree replication strategy: data centers around the world are organized into a tree and writes at the root are replicated downward. As a response to server failure, the tree is reorganized: such reorganization requires all nodes in the tree to be accessing the new data. Therefore, the system must accommodate a consistent data view across all edge machines.

3 Cryptographic Building Blocks

This section describes the various components of the ABE scheme implemented in Portunus. We start by describing the language to specify policies and attributes. Next, we define CP-ABE and its security property, collusion resistance. After that, we delve into pairings, a mathematical operation used to build many ABE schemes. This includes the scheme of our choice, TKN20, which uniquely satisfies all of our requirements. Presented informally and intuitively, we strive to make this complex scheme accessible to a broad audience. We further discuss necessary aspects for achieving strong security guarantees. Finally, we conclude by discussing the software implementation of this scheme and a usage example of the ABE library API.

3.1 Policy Specification Language

In Portunus, the set of attributes assigned to data centers is an injective map from labels to values, both represented as strings, e.g., `country: Japan`. The policies that are enforced on the wrapped private keys are non-monotone Boolean formulas (consisting of AND, OR and NOT operators) over statements that demand that a label has a value, or that it does not have a certain value, e.g., `country: Japan` or `country: not Japan`. Table 1 shows some example policies and corresponding semantics.

For the negations (i.e., NOT operators), we put the NOT operator on the attribute value rather than on the entire attribute. This means that, to satisfy a negation, e.g., `country: not Japan`, the attribute set must have an attribute with the same label, i.e., `country`, and it must differ from the value i.e., `Japan`. In contrast, many schemes put the NOT on the entire attribute, e.g., `not country: Japan` [40]. In these schemes, the attribute set satisfies the negation if it does not contain the attribute `country: Japan`. However, the problem with this type of negation is that attribute sets that do not have any attributes with this label trivially satisfy this negation. This is especially problematic when new labels are added. Then, all previously issued keys automatically satisfy the negation, regardless of whether they may have the negated value or not.

To express and represent policies, we implement a simple language that parses strings from the API and converts them into the structures that are consumed by the ABE scheme (Section 3.2.7). This means the front end of our policy language is composed of Boolean expressions as strings, such as `country: JP or (not region: EU)`, while the back end is a monotonic Boolean circuit consisting of wires and gates.

Monotonic Boolean circuits only include AND and OR gates. In order to handle NOT gates, we assign positive or negative values to the wires. Every NOT gate can be placed directly on a wire because of De Morgan's Law, which allows the conversion of a formula like `not (X and Y)` into `not X or not Y`, and similarly for disjunction.

3.2 Attribute-Based Encryption

Attribute-based encryption (ABE) is a variant of public-key cryptography in which the key pairs are associated with attributes rather than individual users [46]. Unlike traditional public-key encryption, ABE allows users to enforce a more fine-grained access control to the encrypted data [3, 9, 27, 42, 59]. There are two variants of ABE: key-policy ABE (KP-ABE) [27], and ciphertext-policy ABE (CP-ABE) [9].

3.2.1 Key-Policy ABE (KP-ABE)

In KP-ABE, users' secret keys are generated based on an access policy that defines the privileges scope of the concerned user, and data are encrypted over a set of attributes. For example, consider a military setting. A confidential document about nukes is encrypted under the attributes `type: nuclear, clearance: top-secret`. Then a user with a key defined over the access policy `(type: nuclear or type: laser) and clearance: top-secret` can decrypt the document, but a user with a key `clearance: top-secret` cannot.

3.2.2 Ciphertext-Policy ABE (CP-ABE)

In CP-ABE, encrypting users specify access policies that determine who is allowed to decrypt the data. Users' secret keys are generated over a set of attributes. For example, consider a hospital setting in which a doctor has attributes `role: doctor` and `region: US`, while a nurse has attributes `role: nurse` and `region: EU`. A document encrypted under the policy `role: doctor or region: EU` can be decrypted by both the doctor and nurse.

We restrict our discussion to CP-ABE in this paper, because it is a more natural fit to the desired semantics of Portunus: our fleet of servers have natural attributes like location and compliance standards, and our customers choose their policies.

Table 1: Example Policies and Semantics

Example Policy	Semantics
`country: US or region: EU`	Decrypt only in US or European Union
`NOT (country: RU or country: US)`	Do not decrypt in Russia and US
`country: US and security: high`	Decrypt only in US data centers with a high level of security

3.2.3 Formal Definition of CP-ABE

A ciphertext-policy ABE (CP-ABE) scheme consists of four algorithms [9]:

- Setup(λ) $\rightarrow$ (MPK, MSK): The setup takes as input a security parameter λ, it outputs the master public-secret key pair (MPK,MSK).
- KeyGen(MSK,S) $\rightarrow$ SK_S: The key generation takes as input a set of attributes S and the master secret key MSK, and outputs a secret key SK_S.
- Encrypt(MPK, $\mathbb{A}$, M) $\rightarrow$ $CT_{\mathbb{A}}$: The encryption takes as input a plaintext message M, an access policy $\mathbb{A}$ and the master public key MPK. It outputs a ciphertext $CT_{\mathbb{A}}$.
- Decrypt(SK_S,$CT_{\mathbb{A}}$) $\rightarrow M'$: The decryption takes as input the ciphertext $CT_{\mathbb{A}}$ that was encrypted under an access policy $\mathbb{A}$, and a secret key SK_S associated with a set of attributes S. It succeeds and outputs the plaintext message M' if S satisfies $\mathbb{A}$. Otherwise, it aborts.

A scheme is called correct if decryption of a ciphertext with secret key yields the original plaintext message.

3.2.4 Collusion Resistance

The security models for ABE schemes consider their security against chosen-plaintext (CPA) and chosen-ciphertext attacks (CCA), as well as their collusion resistance. Informally, collusion resistance ensures that multiple users with secret keys cannot join forces and decrypt a ciphertext that they could not decrypt individually. For example, a ciphertext encrypted under the policy `role: doctor and region: EU` cannot be decrypted by a user with the attributes `role: doctor` and `region: US`, and another user with the attributes `role: nurse` and `region: EU`. To capture this type of security, the security models allow the attacker to request multiple secret keys for attributes that are not authorized to decrypt the challenge ciphertext. Furthermore, the models capture security against chosen-plaintext attacks or chosen-ciphertext attacks. We define these security models more formally in Appendix A.

3.2.5 Pairing-Based ABE

A popular type of ABE is pairing-based ABE, because it is efficient and can support many desirable properties [59]. A pairing—also known as a bilinear map—is a map $e: \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ defined over three groups $\mathbb{G}_1$,$\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p with generators $g_1 \in \mathbb{G}_1$,$g_2 \in \mathbb{G}_2$ such that (i) $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$ for all $a,b \in \mathbb{Z}_p$ (bilinearity), (ii) $e(g_1, g_2)$ is not the identity in $\mathbb{G}_T$ (non-degeneracy) and (iii) e is efficiently computable. Note that $\mathbb{Z}_p$ denotes the ring of integers modulo p.

Intuitively, pairings are used to ensure that we can achieve security guarantees for both the keys and the ciphertexts. We need those guarantees, because we require ABE schemes to be secure against collusion, meaning that users should not be able to combine their keys and obtain better decryption powers. In contrast, traditional public-key encryption typically only provides security guarantees for the ciphertexts. Therefore, we can use discrete-log based assumptions such as the Diffie-Hellman assumption [19] to create secure encryption schemes such as the ElGamal encryption scheme [26]. In such encryption schemes, the public key and ciphertext typically live in a group in which the discrete-log problem is believed to be hard, while the associated secret key is an integer. By exponentiating a part of the ciphertext with the secret key, we can obtain the message. To ensure that we can achieve similar security assumptions for the keys in ABE, we also place the keys in a group in which the discrete-log problem is believed to be hard. To recover the message, we perform a pairing operation instead of exponentiating, which can be seen as an exponentiation with a "hidden" integer.

Most ABE implementations rely on open-source libraries for the pairing-based arithmetic, e.g., MIRACL [49], RELIC [4] or our own library, CIRCL [1]. In this way, ABE can be implemented in a highly optimized fashion without requiring all the details about the inner workings of pairings. Furthermore, using pairings in a black-box way also allows us to efficiently update the underlying pairing-friendly curves, should the old ones be broken or more efficient ones be found [18].

3.2.6 The TKN20 Scheme

We are using a fully CCA-secure hybrid encryption scheme based on the scheme by Tomida, Kawahara and Nishimaki (TKN20) [54–56]. We have open-sourced this code as part of our cryptographic library, CIRCL [1]. We chose TKN20 because it is currently the only ABE scheme that has a full description and satisfies the following properties simultaneously [59]:

1. **Expressivity:** support for AND, OR and NOT operators. Many schemes exist that support monotone formulas, i.e., formulas with AND and OR only. Few of these also

support NOT operators[1].

2. **(Almost) completely unbounded:** any string can be used as an attribute, and there are no bounds on the policy lengths and attribute sets. Note, however, that it is bounded in the number of label occurrences in the secret key, i.e., each label may occur only once.

3. **Multi-use of attributes:** support for repeated use of the same attribute in a Boolean formula.

4. **Strong security guarantees:** full security against chosen-plaintext attacks under standard assumptions[2].

3.2.7 Representation of Monotone Access Policies

In the mathematical description of the scheme, the (monotone) access policies are represented as linear secret-sharing scheme (LSSS) matrices [28]. In such matrices, the rows of the matrix are associated with the attributes used in the policy. To determine whether a set of attributes S satisfies the policy, the subset of rows associated with the attributes that also occur in the set can be considered. If the vector $(1,0,...,0)$ is in the span of those rows, then the set satisfies the policy matrix.

More formally, an access policy can be represented as a pair $\mathbb{A} = (\mathbf{A}, \rho)$ such that $\mathbf{A} \in \mathbb{Z}_p^{n_1 \times n_2}$ is an LSSS matrix, where $n_1, n_2 \in \mathbb{N}$, and ρ is a function that maps its rows to attribute values. Then, for some vector with randomly generated entries $\mathbf{v} = (s, v_2, ..., v_{n_2}) \in \mathbb{Z}_p^{n_2}$, the i-th share of secret s generated by this matrix is $\lambda_i = \mathbf{A}_i \mathbf{v}^\intercal = A_{i,1} s + \sum_{j \in \{2,...,n_2\}} A_{i,j} v_j$, where $\mathbf{A}_i$ denotes the i-th row of $\mathbf{A}$. In particular, if S satisfies $\mathbb{A}$, then there exist a set of rows $\Upsilon = \{i \in \{1,...,n_1\} \mid \rho(i) \in S\}$ and coefficients $\varepsilon_i \in \mathbb{Z}_p$ for all $i \in \Upsilon$ such that $\sum_{i \in \Upsilon} \varepsilon_i \mathbf{A}_i = (1,0,...,0)$, and by extension $\sum_{i \in \Upsilon} \varepsilon_i \lambda_i = s$, holds.

An efficient method to convert a Boolean formula to an LSSS-matrix representation was proposed by Lewko and Waters [36]. For example, the policy `role: doctor and region: EU` is represented as $(\mathbf{A}, \rho)$ where $\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}$ and ρ maps the first row to `doctor` and the second row to `EU`. The vector $(1,0)$ can only be recovered from both rows, i.e., by adding them. Note that this algorithm yields the same shares of the secret s as the secret-sharing algorithm in the TKN20 paper.

3.2.8 Representing NOTs and Labels

To represent NOT operators and labels in the policy, we define two additional maps, $\bar{\rho}$ and ρ_{lab}. The map $\bar{\rho}: \{1,...,n_1\} \to \{0,1\}$ maps the rows of the matrix (which each correspond to an attribute in the policy) to 0 if the attribute is not negated, and to 1 if the attribute is negated, e.g., `not region: EU`. The map $\rho_{\text{lab}}: \{1,...,n_1\} \to \{0,1\}^*$ maps the rows of the matrix to labels (represented as strings), e.g., `region`.

3.2.9 High-Level Overview of the TKN20 Scheme

Before we give a description of a simplified version of the TKN20 scheme, we first give an overview of the scheme. By doing this, we aim to demystify the many components of the scheme and highlight the techniques used to construct it.

First, we consider the general form of the scheme's master public key, the secret keys and the ciphertexts. In general, the ciphertext consists of one element in $\mathbb{G}_T$ that hides the message, i.e., $M \cdot A^s$, where $A = e(g_1, g_2)^\alpha$ is part of the public key, and further, elements in $\mathbb{G}_1$ and $\mathbb{G}_2$. The secret keys consists of elements in $\mathbb{G}_1$ and $\mathbb{G}_2$, where at least one contains α "in the exponent", e.g., $g_1^{\alpha + rb}$. To decrypt, the appropriate key and ciphertext components need to be paired (with e) to recover $A^s = e(g_1, g_2)^{\alpha s}$, and thus, the message M.

To embed the attribute sets and policies in the secret keys and ciphertexts, we use appropriate representations of these in $\mathbb{G}_1$ and $\mathbb{G}_2$. To represent the policies, we use the shares λ_i generated with the matrix representation in Section 3.2.7. In the scheme, these shares occur as B^{λ_i} in the ciphertext, where B is part of the master public key. To represent the attribute label-value pairs, we use two techniques: the hash-based [28] and the polynomial-based [10] approaches. The hash-based approach simply takes as input the attribute string, e.g., `role: doctor`, and hashes it directly into $\mathbb{G}_1$ or $\mathbb{G}_2$. The polynomial-based approach takes as input the string and first hashes it to an element x in $\mathbb{Z}_p$, and then maps it into $\mathbb{G}_1$ or $\mathbb{G}_2$ with an implicit polynomial, e.g., $B_0 \cdot B_1^x = g_1^{b_0 + x b_1}$. In TKN20, these two approaches are combined: a hash is used to map the attribute-label string directly into the group $\mathbb{G}_1$, and the implicit polynomial is used to map the attribute-value string into the group. More specifically, this combination computes $H_0(\text{lab}) \cdot H_1(\text{lab})^x$, where lab denotes the label, e.g., `role`, and x denotes the representation of the associated value, e.g., `doctor`, in $\mathbb{Z}_p$.

The reason why TKN20 maps the attribute values into the group using the polynomial-based approach is that it can support NOT operators. To support these, TKN20 uses the high-level approach introduced by Ostrovsky et al. [40], which exploits the structure of the polynomial-based map. Roughly, this approach uses the fact that two distinct points on a 1-degree polynomial can be used to reconstruct the polynomial with Lagrange interpolation[3]. More concretely, this means that the secret can be reconstructed if the attribute value in the key does not match the attribute value in the ciphertext, i.e., when they represent two distinct points on the polynomial.

3.2.10 Simplified Description of the TKN20 Scheme

We provide a simplified version of the scheme below, and explain then how the real version of the scheme—which can

[1] NOT operators can be supported in three ways [5]. TKN20 supports the most efficient variant proposed by Okamoto and Takashima [39]. This variant requires that the attribute set uses each label at most once.

[2] We do, however, require the use of the random oracle model [7]

[3] This approach is also used in Shamir's secret sharing scheme [50].

be found in the TKN20 paper [55, 56]—can be constructed from the simplified version.

- Setup(λ) $\rightarrow$ (MPK, MSK): The setup outputs the master public-secret key pair (MPK, MSK), where $H_i\colon \{0,1\}^* \rightarrow \mathbb{G}_1$ with $i \in \{0,1\}$ are two hash functions (modeled as random oracles), MSK $= (\alpha, b)$, and

$$\text{MPK} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, H_0, H_1,$$
$$A = e(g_1, g_2)^{\alpha}, B = g_1^b).$$

- KeyGen(MSK, (S, ψ_{lab})) $\rightarrow \text{SK}_S$: On input a set of attribute values S and the associated labeling map $\psi_{\text{lab}}\colon S \rightarrow \{0,1\}^*$, which maps the attributes in the set S to labels (represented as strings), it outputs the secret key SK_S as

$$\text{SK}_S = (S, K_1 = g_1^{\alpha + rb}, K_2 = g_2^r,$$
$$\{K_{3,\text{att}} = (H_0(\psi_{\text{lab}}(\text{att})) \cdot H_1(\psi_{\text{lab}}(\text{att}))^{x_{\text{att}}})^r\}_{\text{att} \in S}),$$

  where $r \in_R \mathbb{Z}_p$ is a randomly generated element in $\mathbb{Z}_p$ and x_{att} denotes the representation of att in $\mathbb{Z}_p$.

- Encrypt(MPK, $\mathbb{A}, M$) $\rightarrow \text{CT}_{\mathbb{A}}$: On input a plaintext message $M \in \mathbb{G}_T$ and an access policy $\mathbb{A} = (\mathbf{A}, \rho, \rho_{\text{lab}}, \overline{\rho}, \tau)$—where $\tau\colon \{1,\ldots,n_1\} \rightarrow \{1,\ldots,m\}$ is a function that maps each row that is associated with the same label to a different integer in $\{1,\ldots,m\}$, with m being the maximum number of times that a label occurs in the policy—it outputs a ciphertext $\text{CT}_{\mathbb{A}}$ as

$$\text{CT}_{\mathbb{A}} = (\mathbb{A}, C = M \cdot A^s, C_1 = g_2^s, \{C_{2,l} = g_2^{s_l}\}_{l \in \{1,\ldots,m\}},$$
$$\left\{C_{3,j} = B^{\lambda_j} \cdot (H_0(\rho_{\text{lab}}(j)) \cdot H_1(\rho_{\text{lab}}(j))^{x_{\rho(j)}})^{s_{\tau(j)}}\right\}_{j \in \chi_0},$$
$$\left\{C_{3,j} = B^{-\lambda_j} \cdot H_0(\rho_{\text{lab}}(j))^{s_{\tau(j)}},\right.$$
$$\left. C_{4,j} = B^{x_{\rho(j)}\lambda_j} \cdot H_1(\rho_{\text{lab}}(j))^{s_{\tau(j)}}\right\}_{j \in \chi_1}),$$

  where $s, s_1, \ldots, s_m, v_2, \ldots, v_{n_2} \in_R \mathbb{Z}_p$ are randomly generated elements in $\mathbb{Z}_p$, $\lambda_j = A_{j,1}s + \sum_{k \in \{2,\ldots,n_2\}} A_{j,k}v_k$, and $\chi_i = \{j \in \{1,\ldots,n_1\} \mid \overline{\rho}(j) = i\}$ for $i \in \{0,1\}$.

- Decrypt($\text{SK}_S, \text{CT}_{\mathbb{A}}$) $\rightarrow M'$: On input the ciphertext $\text{CT}_{\mathbb{A}}$, and a secret key SK_S, it checks whether S satisfies $\mathbb{A}$. If not, then it aborts. Otherwise, it computes the message by first determining $\Upsilon_0 = \{j \in \chi_0 \mid \rho(j) \in S\}$, $\Upsilon_1 = \{j \in \chi_1 \mid \rho(j) \notin S \wedge \rho_{\text{lab}}(j) \in \psi_{\text{lab}}(S)\}$, $\Upsilon = \Upsilon_0 \cup \Upsilon_1$ and $\{\varepsilon_j\}_{j \in \Upsilon}$ such that $\sum_{j \in \Upsilon} \varepsilon_j \mathbf{A}_j = (1,0,\ldots,0)$, then computing

$$e(g_1, g_2)^{\alpha s} = e(K_1, C_1)$$
$$\cdot \left(\prod_{j \in \Upsilon_0} \left(e(K_{3,\rho(j)}, C_{2,\tau(j)}) / e(C_{3,j}, K_2)\right)\right.$$
$$\left. \cdot \prod_{j \in \Upsilon_1} \left(e(K_{3,\rho(j)}, C_{2,\tau(j)}) / e(C_{3,j}^{y_j} \cdot C_{4,j}, K_2)^{\frac{1}{x_{\rho(j)} - y_j}} \right) \right),$$

  where $y_j = x_{\psi_{\text{lab}}^{-1}(\rho_{\text{lab}}(j))}$. Then, $M = C / e(g_1, g_2)^{\alpha s}$.

3.2.11 Description of the Fully Secure Variant

The structure of the actual TKN20 scheme [55] is much more advanced. This is because the scheme is fully secure under well-studied assumptions, in particular, a variant of the matrix decisional Diffie-Hellman assumption [21]. This assumption is closely related to the decisional Diffie-Hellman assumption [19, 21]. The main technique that is used to achieve this level of security is the dual-system encryption technique [62]. Currently, the most advanced and efficient techniques [16, 35] in this paradigm use matrix structures "in the exponent", e.g., mapping the key component $K_1 = g_1^{\alpha + rb}$ to $g_1^{\mathbf{a} + \mathbf{W}\mathbf{r}}$, where $\mathbf{a}$ and $\mathbf{r}$ are vectors of length 3 and $\mathbf{W}$ is a (3×3)-matrix [35].

3.2.12 Support for Wildcards

To support CCA-security more efficiently than e.g., [64], we use wildcards in the secret keys (as also proposed in the journal version of TKN20, i.e., [56]). A wildcard is represented by an asterisk $*$, e.g., `region: *`, and means that all values for the associated label are accepted, e.g., `region: EU`. In other words, it always matches any occurrence of an attribute with the same label in the policy. The keys for asterisks have the following form:

$$(K_{3,1,\text{att}}, K_{3,2,\text{att}}) = (H_0(\psi_{\text{lab}}(\text{att}))^r, H_1(\psi_{\text{lab}}(\text{att}))^r).$$

Tomida et al. [56] show that the variant of the scheme using wildcards is provably fully secure as well. Note that we use this functionality only to achieve CCA-security, because this functionality seems less intuitive to use for other purposes. In particular, handing out a wildcarded attribute for some label gives the user much power: it always satisfies any occurrence of that specific attribute label in the policy, regardless of what the policy dictates that the user should have.

3.2.13 Key Encapsulation and Symmetric Encryption

We use the TKN20 scheme to encapsulate a symmetric key to be used to encrypt the data, and use a one-time secure symmetric encryption scheme to encapsulate the data. More accurately, we first derive a symmetric key from the ABE ciphertext. In particular, instead of encrypting some message $M \in \mathbb{G}_T$, we directly derive the symmetric key from $e(g_1, g_2)^{\alpha s}$ by applying a key derivation function [17]. Because $e(g_1, g_2)^{\alpha s}$ is indistinguishable from a random element in $\mathbb{G}_T$, the derived key is also indistinguishable from a random key [31, 33]. Then, we use this random key to symmetrically encrypt the data. For this, we use a symmetric encryption scheme that is one-time secure, which means that no attackers can distinguish between the encryptions of any two messages (see Appendix B for a more formal definition). This "hybrid encryption" variant—where we use ABE to encapsulate a key and symmetric encryption to encapsulate the data—is provably secure against chosen-plaintext attacks, see e.g., [34, §A]. To encrypt symmetrically, we use the same approach as Boneh and Katz [12]. We

use a pseudo-random generator to generate a key stream that has the same length as the message, and XOR it to the message. In Portunus, we use an extendable output function to generate a sufficiently-long key stream, i.e., BLAKE2b [45], which is believed to be indistinguishable from pseudo-random generator.

3.2.14 CCA-Security via the BK-Transform

Finally, to achieve CCA-security, we apply the Boneh-Katz transform [12]. With this transform, we combine the hybrid encryption scheme with a message authentication code (MAC) function and a special commitment scheme[4], for which formal definitions and security models can be found in Appendix B. Informally, the special commitment scheme that we use consists of two independent hash functions. The first hash is used to generate a public commitment to a secret random value, and the second hash uses the secret random value to derive a key K'. Subsequently, this secret random value is included in the encryption of the message with the hybrid encryption scheme. We compute a MAC with the key K' over the resulting ciphertext to ensure authenticity of the ciphertext. Furthermore, the public commitment to the secret random value is included in the access policy with an AND operator applied to the original policy, and also in plain in the resulting ciphertext. To decrypt, one first recovers the message and secret random value by decrypting the ciphertext. Then, one verifies whether the public commitment is equal to the hash over the secret random value, and then if the MAC verifies correctly given the key derived from the secret random value. We give a full description of the CCA-secure construction (using the simplified version of TKN20) in Appendix C. It follows from [12] and [60] that this construction is CCA-secure.

3.3 Software Implementation

We implemented our scheme as part of the CIRCL library [1] in Go. The particular instantiation of the pairing-friendly groups $\mathbb{G}_1$,$\mathbb{G}_2$ and $\mathbb{G}_T$ that our implementation uses is the BLS12-381 curve [6, 13]. We generated the code for the arithmetic in $\mathbb{Z}_p$ with the Fiat Cryptography tool [20], which formally verifies the correctness of the produced code. We have also optimized the arithmetic for the groups $\mathbb{G}_1$,$\mathbb{G}_2$ and $\mathbb{G}_T$ through judicious choice of representation. Our implementation uses the fast subgroup checks via Bowe's method [14], which allow us to check whether any given point is in the group, e.g., $\mathbb{G}_1$. To hash into groups, we followed the relevant IETF specification [24]. To optimize the decryption algorithm, we use two common tricks that are often used to speed up computing a product of pairings, i.e., by reordering the computations [42] and by sharing the final step of the pairing operations [29]. Figure 1 presents a reproducible program showing the usage of our code.

[4]Boneh and Katz [12] call this an "encapsulation" scheme, but to distinguish it more clearly from key and data encapsulation, we call it "special commitment scheme" in this paper.

4 Design

Armed with the above scheme, we now must construct services to encrypt customer keys, and make them available to those who should have them. Cloudflare logically has four components. The first is a set of edge machines located in geographically spread and distant data centers. These edge machines run a homogeneous mixture of services that terminate TLS and serve HTTP. The actual signing of TLS handshakes takes place in a system called Gokeyless in all relevant cases.

The second component is a centralized set of services in the control-plane responsible for the API that customers interact with to configure their website. One of these services, the certificate manager, handles all configuration relating to TLS.

The third component is a small number of very tightly controlled machines that handle certificate issuance for internal certificates. All machines at Cloudflare have a machine identity based on RSA keys: our key issuance service uses that identity to determine the attributes a machine shall have. We call this service the Key Generation Authority (KGA).

The fourth component is a globally synchronized key-value store, Quicksilver. This is a global gossip tree for customer configurations, such as certificates, that is designed to ensure extremely fast replication, at the cost of constrained bandwidth and storage. Every edge machine stores a local copy of the data in Quicksilver.

4.1 Encrypting Customer Keys

When a customer uploads a certificate and the associated private signing key to Cloudflare, and indicates it is to be protected under an access policy, the certificate manager in the control-plane takes the private key and encrypts it with the required policy. However, the customer's private key is not encrypted with the ABE master public key directly. Rather, it is encrypted with an X25519 key pair, the private key of which is encrypted under the ABE scheme. These key pairs are indexed by the policy and the epoch they are under. At any time, there may be several of these key pairs, called *policy keys*, present in the database for a given policy. The certificate manager will use the most recent one for encryption. This permits gradual rotation of the key pairs. Note that the only encryption happening in Portunus is done by the certificate manager.

4.2 Accessing Customer Keys

On receipt of a connection to a site, such as `alice.test.com`, Gokeyless carries out a lookup for the certificate in Quicksilver. If that certificate has a key protected by Portunus, the metadata for that certificate will have a pointer to the relevant policy key together with a ciphertext that decrypts to the private key. Gokeyless then loads the policy key and determines if it is decryptable by the machine. If not, it consults a table that maps each policy to a list of satisfying data centers to find a

Listing 1: Example usage of the ABE library API

```
masterPubKey, masterSecKey := Setup() // Initialize the master public and master secret key
accessPolicy := new(Policy)
accessPolicy.FromString("country: US or region: EU") // Create new policy from given string
// Encrypt the secret message using the master public key and policy
encryptedMsg := masterPubKey.Encrypt(accessPolicy, []byte{"long live ABE"})
parisDCAttributes := new(Attributes) // Create attributes for the Paris data center
parisDCAttributes.FromMap(map[string]string{ "country": "FR", "region": "EU"})
// Generate an attribute secret key for the Paris data center using the master secret key
parisDCSecKey := masterSecKey.KeyGen(parisDCAttributes)
// Decrypt the ciphertext using the attribute secret key of the Paris data center
decryptedMsg := parisDCSecKey.Decrypt(encryptedMsg)
assertEquals(decryptedMsg, []byte{"long live ABE"})
```

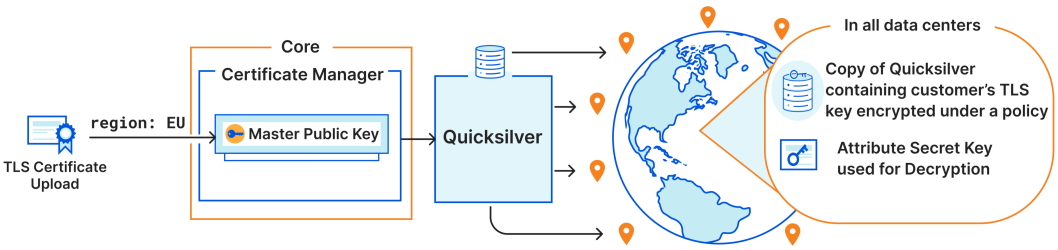


Figure 1: Encryption under a policy

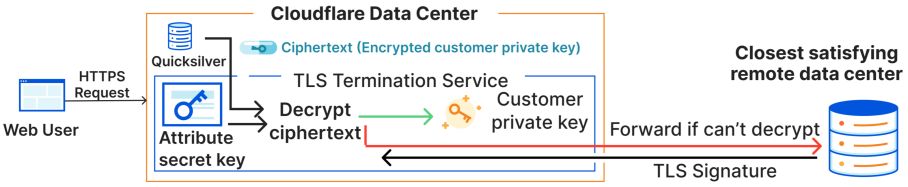


Figure 2: Decryption using Attribute Secret Key

neighboring one, and forwards the request there. Gokeyless on this machine then decrypts the policy key and uses the result to decrypt the certificate's private key, performing the signature and completing the TLS handshake. The decrypted policy keys are cached in memory, so the computationally burdensome ABE decryption only happens once for commonly used policies. This is an important optimization to avoid excessive CPU consumption during attack scenarios when many handshakes are arriving.

4.3 Key Distribution

The key generation authority (KGA) holds the ABE master secret key. It also has access to the unique cryptographic identity for every machine in the fleet, as well as a map of machines to attributes. This map is largely synchronized with the machine's own view. Key issuance for the machine's attribute-based secret key is managed by the service configuration management system, Salt [2]. Salt uses the RSA identity key of the machine to authenticate to the CA, which generates the machine's attribute secret key using the master secret key and the attributes of the machine. The map of machines to attributes is configured in the same database that drives machine identity for Salt.

4.4 Key Rotation

Over time, it is necessary to change the key material in the system so that an attacker who has access to old key material can no longer decrypt newly uploaded customer TLS private keys. However, the lifetime of a customer certificate can extend beyond a rotation period and it must be possible to continue to decrypt the customer TLS key for that duration.

The key generation authority generates a new *generation* of the master key pair. To preserve the ability to decrypt old TLS private keys, the CA re-encrypts the existing policy keys on behalf of the certificate manager. During this process,

machines will have both the old and new generation of the attribute secret key, ensuring that availability is not impacted as the old key material is phased out.

Newly uploaded certificate private keys are encrypted under the same policy key. This means that an attacker who has access to a policy key can continue to to decrypt new TLS keys, but it is possible to generate new policy keys for a policy. This does however guard against an attacker who obtains the attribute secret keys for a machine from being able to access the TLS keys after rotation.

To detect accidental or malicious usage of expired key generations and have end-to-end visibility into the status of key rotation, we have logging and metrics for key generation held and used in each part of the system.

4.5 Attribute Changes

From time to time, the attributes associated with a set of data centers may change. Introducing new labels that have not been used by existing policies is straightforward, since the set of data centers that can decrypt a given TLS private key remains unchanged. However, when the attributes of the data centers that can decrypt a key are changing, certain changes are needed to maintain system functionality [5, 59]. We split the act of "changing an attribute" in two steps: removal of the old label and value, and the re-addition of the label with the new value. The former results in loss of decryption capabilities for associated data centers, because they no longer satisfy the policies that required the presence of this label. Note that the removal does not increase the decryption capabilities yet, because to satisfy a negated attribute, the set of attributes of the data center must have an attribute with the same label, regardless of the value. Adding a new label can only increase the number of policies satisfied due to negation semantics.

Carrying out this transition requires three steps. First, the affected label is removed from the forwarding information of the involved data center, so that other data centers stop sending requests that require its presence. Second, the key is re-issued with the new attribute. Third, the new attribute is re-added to the forwarding information so requests are handled by the data center again. Throughout this, the affected data center handles end-user requests as usual: those requests that cannot be satisfied locally are forwarded to other data centers that can satisfy them, whose forwarding information is not affected by the transition. This process can be difficult to carry out at scale and requires careful planning and should be done in stages. Lastly, a key rotation is required to ensure that any retained copies of the older key are not used.

4.6 Networking and Resiliency

Gokeyless uses an RPC protocol to forward TLS signing requests to the closest satisfying data center, which on arrival leverage a network layer load balancer [63] to determine an appropriate machine to handle the connection. Since the computational load of handling a request forwarded for Portunus is merely an X25519 decryption and an RSA/ECDSA signature, even high levels of request volume have not led to failures due to load balancing issues.

Because maintaining connections to all other machines can be expensive, machines within one data center will elect among themselves a machine to forward requests to specific close foreign data centers. This reduces the number of TCP connections being used.

Resiliency is negatively impacted for customers who apply overly restrictive policies. It is possible for data centers to be taken offline or become overwhelmed for a variety of reasons. If all data centers a customer's key is decryptable in are offline, then the customer's website will be rendered inaccessible. To prevent this, we require customer keys be decryptable in at least two large-capacity data centers.

We have found that typically customers will store their certificates in regions where the majority of their users are found. This unsurprising pattern puts low demands on the remote execution capabilities. Unfortunately, events such as DDoS attacks can add significant load. Operation under normal conditions is not a guide to operation under adverse conditions. This led us to expend significant effort to integrate distributed tracing in addition to metrics to track system performance and quickly diagnose and reproduce scaling issues.

5 Evaluation

While Portunus was launched to customers in 2022, the older version of the system based on similar principles (Geo Key Manager) has been in production since 2017. Over the years, the number of customers and end-users relying on this product has steadily increased. This section is an evaluation of the various components of Portunus.

During a sample week in December 2022, we observed 100k requests per second being served between Portunus and Geo Key Manager. As most customers restrict key access to the region where they typically have the most users, approximately 80% of these requests are handled locally. The remaining 20% are forwarded to their closest satisfying neighbor.

5.1 Cryptography

We evaluate our underlying cryptography library against RSA-2048 and X25519, utilizing Go libraries `crypto/rsa` and `x/crypto/nacl/box` as reference implementations. These comparative algorithms were chosen because they are standard public-key cryptography. We conduct our measurements on an Apple M1 Mac.

We characterize our library's performance using measures inspired by ECRYPT [8]. In all comparisons involving ABE, we set the attribute set size to 50 and consider policy formulas over 50 attributes. This attribute set size is significantly higher

Table 2: Space Overheads (bytes)

Scheme	Secret key[5]	Public key	Encrypt 23 B	Encrypt 10 KB
RSA-2048	1190	256	233	496
X25519	32	32	48	48
Our scheme	23546	3282	19475	19475

Table 3: Operation times (ms)

Scheme	Key Gen.	Encrypt 23 B	Decrypt 23 B
RSA-2048	180	0.209	1.47
X25519	0.061	0.096	0.046
Our scheme	701	364	30.1

than necessary for any of Portunus' applications, as most policies are typically limited to a combination of geographic properties. Nevertheless, it serves as an extreme worst-case scenario for benchmarking purposes.

Table 2 shows the space consumed by the various operations. For our system, the ciphertext overhead is of particular concern since it is replicated on every machine. Unfortunately, this overhead is significantly larger than in traditional public-key cryptography. However, the good news is that this overhead is constant with respect to message length for a given policy size, and can be reduced by relying on a small handful of policy keys (defined in Section 4.1) rather than encrypting every customer key using ABE. Importantly, the ciphertexts in our system can be decrypted by multiple decryptors, whereas standard public-key cryptography benchmarks only consider a ciphertext that can be decrypted by a single decryptor. The size of the attribute secret key is less relevant, as a single copy is stored per machine. The size of the master public key is of even less concern, as it is only used by the certificate manager in the control plane.

Table 3 shows the average time required to perform different key operations. Key generation refers to the process of generating attribute secret keys from the master secret key, which can be performed out-of-band of user handshakes and is therefore of marginal relevance in this context. Encryption latency can also largely be ignored, as it is acceptable for encryption to take a few extra cycles before a certificate is considered deployed. But once it is deployed, HTTPS requests to the website should complete quickly. Since decryption is in the critical path of every request, it is the most pertinent in our situation. While session resumption and caching policy keys can amortize the number of ABE decryptions across TLS handshakes to a small fraction, improvements to decryption latency will still affect overall baseline performance. It is therefore important to further optimize the decryption process.

[5]For ABE, this is the Attribute-Based Secret Key.

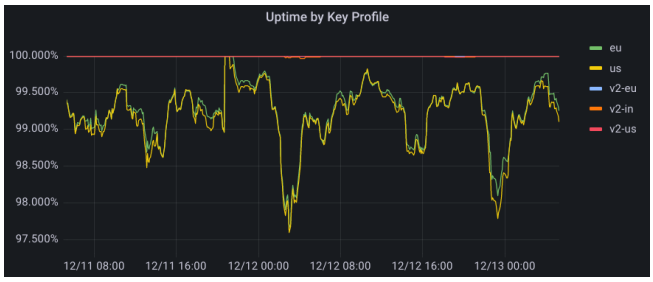


Figure 3: Uptime by policy; this shows that Portunus (v2) has consistently better uptime than Geo Key Manager (v1)

5.1.1 Request Latency

The overall performance of Portunus includes the impact of cryptography, networking and geographic location based on the type of Portunus request: handshakes processed locally, and those forwarded to a remote data center. The vast majority of local requests only perform an X25519 decryption because of policy keys. The remainder incur the overhead of an ABE decryption. For remote requests, network latency largely dominates.

5.2 Availability

Figure 3 shows the uptime of our system by policy vs the previous system. This graph was produced using synthetic probes spanning every machine across our fleet. It demonstrates that dynamically selecting all possible machines to decrypt rather than a pre-determined handful as in our previous release, produces significant improvements to real-world reliability.

6 Discussion

We want to reiterate why an access-control solution based on novel cryptography makes the most sense for our system.

The TLS key management system is responsible for ensuring that edge machines can access customer signing keys when customers upload their TLS certificates (and associated signing keys) to Cloudflare. To ensure availability and low latency, the key manager relies on an internal globally synchronized key-value store, Quicksilver [43] to distribute user configurations to all edge data centers within seconds. A copy of the entire Quicksilver data set is replicated on every edge machine for fault tolerance, allowing requests to be served even if disconnected from the central synchronizing server.

However, augmenting the key management system to support policy-based access restrictions required us to rethink our approach of storing the same data on all edge machines. Using a central server to enforce access would reduce fault tolerance and add additional latency, undermining the advantages of a distributed edge. Alternatively, we could ensure that only data centers that satisfy a given policy receive those policy-restricted keys. However, this would require

modifying Quicksilver's replication strategy to store only a subset of the key set, which challenges core design decisions Cloudflare has made over the years that assume the entire data set is replicated on every machine.

We considered a third option of issuing unique keys for each data center: wrapping each TLS signing key with the key of every authorized data center and adding them all to Quicksilver. Although this approach would have permitted access to the key only in certain locations while letting TLS be terminated where possible, it would also significantly increase the storage space requirements on every machine proportional to the number of datacenters.

This encouraged us to explore alternative cryptographic solutions. Our first attempt, Geo Key Manager, was developed back in 2016, when there was only one ABE scheme that supported all properties [59], but in a rather inefficient way. In particular, at the time, supporting negations led to significant efficiency penalties in the decryption algorithm [58]. Only recently, the community started addressing these efficiency issues [5, 55, 58]. To get around this limitation, we initially used a combination of identity-based encryption and broadcast encryption to simulate an ABE-like scheme. Unfortunately, this scheme was not collusion resistant (Section 3.2.4). As a result we were eager to switch to a more theoretically satisfying solution once practical ABE schemes became available.

During the implemention of our ABE scheme, while we performed much optimization (Section 3.3), the implementation has larger overheads compared to the mature implementations for traditional cryptographic schemes. We mitigated some of these costs by using policy keys (Section 4.1). This approach is similar to hybrid encryption, where public-key cryptography is used to establish a shared symmetric key used to encrypt data. Policy keys are public key encryption keys to permit the central service to encrypt user's certificates without access to the key. While not as efficient as symmetric cryptography this still reduces the overhead.

The performance and reliability improvements of the deployed system are due to side-effects of the new cryptographic scheme. The original system only supported one kind of attribute, a region. Unfortunately, this did not support customers who wanted to specify countries, and so a hard-coded list of cities was used on upload. This list was rarely updated, so new datacenters were not used. Migration was an extremely difficult prospect. Switching to the new system meant that country could be used directly, and additional attributes could be added. This immediately increased the available set of data centers for many common policies, and directly improved reliability.

Although ABE employs a highly trusted key generation authority to issue the secret keys, we argue that this authority does not need to be more trusted than an authority enforcing traditional access control. Specifically, in Portunus, the role of the key generation authority is integrated with a certificate authority that is used to secure all critical services within Cloudflare's internal network. If this authority were to be corrupted, the consequences would be much worse than simply breaking the security of ABE. If, however, a similar setting would require that the trust in the authority is mitigated by distributing the trust across multiple authorities, one could also deploy multi-authority ABE [15, 37].

A policy conundrum may also arise in certain situations: the encrypted data still resides in restricted regions. This can potentially cause concern among those without a comprehensive understanding of the system. Assuaging these concerns will vary between organizations, but a big part involves spreading awareness of how policy-based encryption works.

7 Future Work

Although ABE can support all properties required by our particular application, it does present minor limitations that may be critical in other contexts. For instance, our scheme doesn't support policies with wildcards of the form `country: *`, meaning any country can satisfy this policy. It likewise doesn't permit an attribute set with multiple values for a single attribute label, such as `{group: fiddlers, group: percussionists}`.

It is unclear what post-quantum ABE schemes will have the combination of performance, implementation simplicity, and expressivity required. Likewise, the use of pairing-based cryptography creates some challenges in acceptance, as decision makers may be unfamiliar with it and it is not standardized, despite ongoing efforts towards standardization at IETF [47].

8 Lessons Learned

In the course of operating Portunus and its predecessor Geo Key Manager, we have learnt several lessons.

Even after more practical ABE schemes became available, the difficulty in translating a scheme from an ABE paper to practice, as well as in selecting an appropriate scheme, should not be underestimated. Typically, there are some parameters that must be chosen, with little indication of the strength of the various assumptions the parameters create. In addition, the notation can require a formidable amount of translation, sometimes concealing significant computational steps.

There persists a prevalent notion in the cryptography community that ABE is still unreasonably slow to be useful. We believe this is no longer true. Just like traditional public key cryptography is not used independently to encrypt large amounts of data, but rather in concert with symmetric encryption, we believe many applications can enjoy the benefits of ABE using it with a hybrid encryption strategy.

Complicated cryptographic schemes in services, particularly ones as critical as TLS termination, can elicit operational apprehensions amongst SREs and other teams that depend on the service. Cryptography can end up being scapegoated when issues arise, despite the problems originating from other system components. We believe when designing such a system, it is prudent

to prioritize simplicity in every other aspect. This makes failures outside the cryptography straightforward to diagnose and conserves the complexity budget for the cryptography.

We have certainly faced the consequences of not adhering to this principle, in the form of delayed rollout due to difficulty garnering operational confidence. Geo Key Manager's complexity extended beyond just the cryptographic components, such as the use of a custom RPC protocol - an artifact of the parent system's (which Geo Key Manager was integrated into) development before the existence of gRPC. This custom protocol, despite resolving most of its quirks as part of Portunus's development, continued to present challenges in specific edge cases. We are presently transitioning to gRPC to alleviate these issues and preempt future related issues. Another example of reducing complexity was replacing a complex thread-pool architecture based on outdated assumptions, with a simple and scalable architecture of one goroutine per request, capitalizing on the lightweight concurrency model offered by Go. Our ongoing efforts aim to simplify various other components, with the ultimate goal of improving system maintainability and fostering enthusiastic buy-in from other stakeholders, as well as encouraging other teams to consider applying ABE-based access control for their own use cases, armed with the reassurance of Portunus's successful deployment.

9 Related Work

Prior work has considered CP-ABE for enforcing access control. Oftentimes, the design of new ABE schemes is justified by their new capabilities based on use in access control [9,44,65], but without many details on potential system design. However, as Venema and Alpár demonstrate [57], there have been various attacks on some of these constructions [44,65], particularly those that do not rely on pairings [30,66]. It is therefore important that care is taken in choosing an established scheme that has the necessary properties and is secure.

Although there are many schemes with various properties [59], our chosen scheme is unique in satisfying our desired properties 3.2.6, making comparisons with alternative ABE schemes difficult. Likewise, comparing cryptographic operation benchmarks with potentially sub-optimized research implementations of ABE schemes is not in scope and we refer the reader to more comprehensive analyses [18].

In contrast to most works about systems that use ABE, Sieve [61] dives deeper into the details of the system. The authors discuss how key management, ABE overhead, and key deletion have to be addressed in a deployed system. In this system, applications can interact with user data stored in the cloud, while users maintain control over which applications have access to their data. Because Sieve is built for a different setting than Portunus, it makes different choices in how ABE is applied. Most notably, it uses key-policy ABE, where the keys are associated with policies and the ciphertexts are associated with attribute sets, to apply a "tag-based" access control like described in Section 3.2.1. In particular, the encrypted data objects that are stored in the cloud are associated with attribute sets. Once an application requests access to the user's data, the user can specify a policy stating which data can be accessed.

Excalibur [48] is perhaps the most similar to our work, because it also uses CP-ABE to enforce access control in the spirit of attribute-based access control. The authors designed and implemented a system for customers to make data accessible on certain machines, and integrated it into a cloud environment. Because their use case is different, the challenges that their system overcomes also differ. Importantly, like Sieve, Excalibur did not progress beyond a lab setting and was never actually deployed in a large-scale real-world setting.

Finally, most existing access-control applications separate the cryptographic and access-control aspects. Cryptography is used to protect sensitive data (in transit, but sometimes, also at rest), while access control is enforced more traditionally. In traditional models, a central authority typically enforces policies by validating an entity's authorization prior to granting data access. When access is granted, a decryption key is shared with the requesting entity, who can then access the data by decrypting it. However, this approach introduces significant latency to the access request. Furthermore, a single central authority makes the system vulnerable to denial-of-service attacks. In contrast, using ABE allows access control on a cryptographic level, achieving both protection of the data whilst allowing the enforcement of access control. By extension, it removes this extra latency and mitigates the availability issues of the central authority. It also allows data to be freely transmitted between nodes, removing the risk of accidental data leakage as long as the target node's secret key is not compromised.

10 Conclusion

For several months, Portunus has seen real-world usage protecting customer keys. It has succeeded in supplanting the capabilities of the legacy system and setting a foundation for future product development. This required multiple person-years of effort by a small team, as well as accepting a fairly novel scheme as the foundation of its security. This effort brought about increases in reliability and enhanced performance.

Acknowledgements

We are grateful for the assistance of the Security Engineering team, particularly Mihir Jham and Nicky Semenza, in explaining their software and letting us make some rather audacious proposals to redesign a significant system. We would also like to thank the Research team. Finally, Mike Rosulek helped us solve the hardest problem in Computer Science by suggesting the name Portunus[6].

[6]The Roman God of keys, thresholds, gates and ports.

References

[1] Cloudflare interoperable, reusable cryptographic library. https://pkg.go.dev/github.com/cloudflare/circl@v1.3.0/abe/cpabe/tkn20. Accessed: 2022-12-14.

[2] Salt project documentation. https://docs.saltproject.io/en/latest/. Accessed: 2022-10-25.

[3] Joseph A. Akinyele, Matthew W. Pagano, Matthew D. Green, Christoph U. Lehmann, Zachary N. J. Peterson, and Aviel D. Rubin. Securing electronic medical records using attribute-based encryption on mobile devices. In Xuxian Jiang, Amiya Bhattacharya, Partha Dasgupta, and William Enck, editors, *SPSM'11*, pages 75–86. ACM, 2011.

[4] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[5] Nuttapong Attrapadung and Junichi Tomida. Unbounded dynamic predicate compositions in ABE from standard assumptions. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT*, volume 12493 of *LNCS*, pages 405–436. Springer, 2020.

[6] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN*, volume 2576 of *LNCS*, pages 257–267. Springer, 2002.

[7] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS*, pages 62–73. ACM, 1993.

[8] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to/results-encrypt.html. Accessed: 2022-10-25.

[9] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *S&P*, pages 321–334. IEEE, 2007.

[10] Dan Boneh and Xavier Boyen. Efficient selective-id secure identity-based encryption without random oracles. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *LNCS*, pages 223–238. Springer, 2004.

[11] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In Joe Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.

[12] Dan Boneh and Jonathan Katz. Improved efficiency for cca-secure cryptosystems built using identity-based encryption. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 87–103, 2005. https://www.cs.umd.edu/~jkatz/papers/id-cca-mac.pdf.

[13] S. Bowe. BLS12-381: New zk-SNARK elliptic curve construction. https://blog.z.cash/new-snark-curve/.

[14] Sean Bowe. Faster subgroup checks for bls12-381. Cryptology ePrint Archive, Paper 2019/814, 2019. https://eprint.iacr.org/2019/814.

[15] Melissa Chase. Multi-authority attribute based encryption. In Salil P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 515–534. Springer, 2007.

[16] Jie Chen, Romain Gay, and Hoeteck Wee. Improved dual system ABE in prime-order groups via predicate encodings. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT*, volume 9057 of *LNCS*, pages 595–624. Springer, 2015.

[17] Lily Chen. Recommendation for key derivation using pseudorandom functions. Technical Report NIST Special Publication (SP) 800-108, Rev. 1, National Institute of Standards and Technology, Gaithersburg, MD, 2022.

[18] Antonio de la Piedra, Marloes Venema, and Greg Alpár. ABE squared: Accurately benchmarking efficiency of attribute-based encryption. *TCHES*, 2022(2):192–239, 2022.

[19] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[20] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1202–1219. IEEE, 2019.

[21] Alex Escala, Gottfried Herold, Eike Kiltz, Carla Ràfols, and Jorge L. Villar. An algebraic framework for diffie-hellman assumptions. In Ran Canetti and Juan A. Garay, editors, *CRYPTO*, volume 8043 of *Lecture Notes in Computer Science*, pages 129–147. Springer, 2013.

[22] ETSI. ETSI TS 103 458 (V1.1.1). Technical specification, European Telecommunications Standards Institute (ETSI), 2018.

[23] ETSI. ETSI TS 103 532 (V1.1.1). Technical specification, European Telecommunications Standards Institute (ETSI), 2018.

[24] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to Elliptic Curves. RFC 9380, June 2023. https://doi.org/10.17487/rfc9380.

[25] Amos Fiat and Moni Naor. Broadcast encryption. In Douglas R. Stinson, editor, *CRYPTO*, volume 773 of *Lecture Notes in Computer Science*, pages 480–491. Springer, 1993.

[26] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *CRYPTO*, volume 196 of *LNCS*, pages 10–18. Springer, 1984.

[27] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *CCS*, pages 89–98. ACM, 2006.

[28] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. Cryptology ePrint Archive, Paper 2006/309, 2006. https://eprint.iacr.org/2006/309.

[29] R Granger and N. P. Smart. On computing products of pairings. Cryptology ePrint Archive, Paper 2006/172, 2006. https://eprint.iacr.org/2006/172.

[30] Javier Herranz. Attacking pairing-free attribute-based encryption schemes. *IEEE Access*, 8:222226–222232, 2020.

[31] Susan Hohenberger and Brent Waters. Online/offline attribute-based encryption. In Hugo Krawczyk, editor, *PKC*, volume 8383 of *LNCS*, pages 293–310. Springer, 2014.

[32] Seny Kamara and Kristin E. Lauter. Cryptographic cloud storage. In Radu Sion, Reza Curtmola, Sven Dietrich, Aggelos Kiayias, Josep M. Miret, Kazue Sako, and Francesc Sebé, editors, *FC*, volume 6054 of *LNCS*, pages 136–149. Springer, 2010.

[33] Eike Kiltz and Yevgeniy Vahlis. CCA2 secure IBE: standard model efficiency through authenticated symmetric encryption. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *LNCS*, pages 221–238. Springer, 2008.

[34] Eike Kiltz and Yevgeniy Vahlis. Cca2 secure ibe: Standard model efficiency through authenticated symmetric encryption. Cryptology ePrint Archive, Paper 2008/020, 2008. https://eprint.iacr.org/2008/020.

[35] Lucas Kowalczyk and Hoeteck Wee. Compact adaptively secure ABE for \mathsf nc^1 from k-lin. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT*, volume 11476 of *LNCS*, pages 3–33. Springer, 2019.

[36] Allison Lewko and Brent Waters. Decentralizing attribute-based encryption. Cryptology ePrint Archive, Paper 2010/351, 2010. https://eprint.iacr.org/2010/351.

[37] Allison Lewko and Brent Waters. Decentralizing attribute-based encryption. In *EUROCRYPT*, pages 568–588. Springer, 2011.

[38] Vincent Hu (NIST), David Ferraiolo (NIST), Richard Kuhn (NIST), Adam Schnitzer (BAH), Kenneth Sandlin (MITRE), Robert Miller (MITRE), and Karen Scarfone (Scarfone Cybersecurity). Guide to attribute based access control (abac) definition and considerations. Technical report, National Institute of Standards and Technology, 2014. https://doi.org/10.6028/NIST.SP.800-162.

[39] Tatsuaki Okamoto and Katsuyuki Takashima. Fully secure functional encryption with general relations from the decisional linear assumption. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *LNCS*, pages 191–208. Springer, 2010.

[40] Rafail Ostrovsky, Amit Sahai, and Brent Waters. Attribute-based encryption with non-monotonic access structures. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *CCS*, pages 195–203. ACM, 2007.

[41] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christina D. Richards, and Mengzhi Wang. Zanzibar: Google's consistent, global authorization system. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, Renton, WA, 2019. https://research.google/pubs/pub48190/.

[42] Matthew Pirretti, Patrick Traynor, Patrick D. McDaniel, and Brent Waters. Secure attribute-based systems. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *CCS*, pages 99–112. ACM, 2006.

[43] Geoffrey Plouviez. Introducing quicksilver: Configuration distribution at internet scale.

https://blog.cloudflare.com/introducing-quicksilver-configuration-distribution-at-internet-scale, 2020. Accessed: 2022-10-25.

[44] Huiling Qian, Jiguo Li, Yichen Zhang, and Jinguang Han. Privacy-preserving personal health record using multi-authority attribute-based encryption with revocation. *Int. J. Inf. Secur.*, 14(6):487–497, nov 2015.

[45] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 cryptographic hash and message authentication code (MAC). Technical Report 7693, 2015. https://www.rfc-editor.org/rfc/rfc7693.

[46] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In R. Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 457–473. Springer, 2005.

[47] Yumi Sakemi, Tetsutaro Kobayashi, Tsunekazu Saito, and Riad S. Wahby. Pairing-Friendly Curves. Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-11, Internet Engineering Task Force, November 2022. Work in Progress.

[48] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In Tadayoshi Kohno, editor, *USENIX Security Symposium*, pages 175–188. USENIX Association, 2012.

[49] Michael Scott. MIRACL cryptographic SDK: Multi-precision Integer and Rational Arithmetic Cryptographic Library. https://github.com/miracl/MIRACL, 2003.

[50] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[51] Adi Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and David Chaum, editors, *CRYPTO*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer, 1984.

[52] Nick Sullivan. Geo key manager: How it works. https://blog.cloudflare.com/geo-key-manager-how-it-works/, 2017. Accessed: 2022-10-25.

[53] Nick Sullivan and Brendan McMillion. Geo key manager. *Real World Crypto 2018*, jan 2018. https://rwc.iacr.org/2018/Slides/Sullivan.pdf.

[54] Junichi Tomida, Yuto Kawahara, and Ryo Nishimaki. Fast, compact, and expressive attribute-based encryption. Cryptology ePrint Archive, Paper 2019/966, 2019. https://eprint.iacr.org/2019/966.

[55] Junichi Tomida, Yuto Kawahara, and Ryo Nishimaki. Fast, compact, and expressive attribute-based encryption. In A. Kiayias, M. Kohlweiss, P. Wallden, and V. Zikas, editors, *PKC*, volume 12110 of *LNCS*, pages 3–33. Springer, 2020.

[56] Junichi Tomida, Yuto Kawahara, and Ryo Nishimaki. Fast, compact, and expressive attribute-based encryption. *Des. Codes Cryptogr.*, 89(11):2577–2626, 2021.

[57] Marloes Venema and Greg Alpár. A bunch of broken schemes: A simple yet powerful linear approach to analyzing security of attribute-based encryption. In Kenneth G. Paterson, editor, *CT-RSA*, volume 12704 of *LNCS*, pages 100–125. Springer, 2021.

[58] Marloes Venema and Greg Alpár. GLUE: generalizing unbounded attribute-based encryption for flexible efficiency trade-offs. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *Public-Key Cryptography - PKC 2023 - 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 7-10, 2023, Proceedings, Part I*, volume 13940 of *Lecture Notes in Computer Science*, pages 652–682. Springer, 2023.

[59] Marloes Venema, Greg Alpár, and Jaap-Henk Hoepman. Systematizing core properties of pairing-based attribute-based encryption to uncover remaining challenges in enforcing access control in practice. *Des. Codes Cryptogr.*, 91(1):165–220, 2023.

[60] Marloes Venema and Leon Botros. Efficient and generic transformations for chosen-ciphertext secure predicate encryption. Cryptology ePrint Archive, Paper 2022/1436, 2022.

[61] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 611–626, Santa Clara, CA, March 2016. USENIX Association.

[62] Brent Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 619–636. Springer, 2009.

[63] David Wragg. Unimog —- cloudflare's edge load balancer. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/, 2020. Accessed: 2022-12-15.

[64] Shota Yamada, Nuttapong Attrapadung, Goichiro Hanaoka, and Noboru Kunihiro. Generic constructions for chosen-ciphertext secure attribute based encryption.

In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC*, volume 6571 of *LNCS*, pages 71–89. Springer, 2011.

[65] Kan Yang, Xiaohua Jia, Kui Ren, Bo Zhang, and Ruitao Xie. DAC-MACS: effective data access control for multiauthority cloud storage systems. *IEEE Trans. Inf. Forensics Secur.*, 8(11):1790–1801, 2013.

[66] Xuanxia Yao, Zhi Chen, and Ye Tian. A lightweight attribute-based encryption scheme for the internet of things. *Future Gener. Comput. Syst.*, 49:104–112, 2015.

A Security Model for CP-ABE

We define the security game IND-CCA(λ) between challenger and attacker as follows:

- **Setup phase:** The challenger runs Setup(λ) to obtain MPK and MSK, and sends the master public key MPK to the attacker.
- **First query phase:** The attacker can make two types of queries:
  - **Key query:** The attacker queries secret keys for sets of attributes S, and obtains $\mathrm{SK}_S \leftarrow \mathrm{KeyGen}(\mathrm{MSK}, S)$ in response.
  - **Decryption query:** The attacker sends a ciphertext $\mathrm{CT}_{\mathbb{A}}$ for access policy $\mathbb{A}$ and some set S that satisfies $\mathbb{A}$ to the challenger, who returns the message $M \leftarrow \mathrm{Decrypt}(\mathrm{MPK}, \mathrm{SK}_S, \mathrm{CT}_{\mathbb{A}})$ (where $\mathrm{SK}_S \leftarrow \mathrm{KeyGen}(\mathrm{MSK}, S)$).
- **Challenge phase:** The attacker specifies some access policy $\mathbb{A}^*$ such that none of the sets S in the first key query phase satisfies $\mathbb{A}^*$, generates two equal-length messages M_0 and M_1, and sends these to the challenger. The challenger flips a coin, i.e., $\beta \in_R \{0,1\}$, encrypts M_β under $\mathbb{A}^*$, i.e., $\mathrm{CT}_{\mathbb{A}^*} \leftarrow \mathrm{Encrypt}(\mathrm{MPK}, \mathbb{A}^*, M_\beta)$, and sends the resulting ciphertext $\mathrm{CT}^*_{\mathbb{A}^*}$ to the attacker.
- **Second query phase:** This phase is identical to the first query phase, with the additional restriction that the attacker cannot query keys for sets of attributes S that satisfy the policy $\mathbb{A}^*$ or make a decryption query for $\mathrm{CT}^*_{\mathbb{A}^*}$.
- **Decision phase:** The attacker outputs a guess β' for β.

The advantage of the attacker is defined as

$$\mathsf{Adv}_{\text{IND-CCA}} = \left| \Pr[\beta' = \beta] - \frac{1}{2} \right|.$$

A scheme is fully secure against chosen-ciphertext attacks if all polynomial-time attackers have at most a negligible advantage in this security game.

In the model for security against chosen-plaintext attacks, the attacker is not allowed to make decryption queries in the first and second query phase—only key queries.

B Other Definitions

B.1 Symmetric Encryption

B.1.1 Formal Definition

We define symmetric encryption as follows. Let λ be the security parameter. A symmetric encryption scheme $\mathrm{SE} = (\mathrm{Enc}, \mathrm{Dec})$, with symmetric key $\mathrm{K} \in \{0,1\}^\lambda$, is defined as

- $\mathrm{Enc}_\mathrm{K}(M)$: On input message $M \in \{0,1\}^*$, encryption returns a ciphertext $\mathrm{CT}_{\mathrm{sym}}$.
- $\mathrm{Dec}_\mathrm{K}(\mathrm{CT}_{\mathrm{sym}})$: On input ciphertext $\mathrm{CT}_{\mathrm{sym}}$, decryption returns a message M or an error message $\perp$.

The scheme is correct if for all keys $\mathrm{K} \in \{0,1\}^\lambda$ and all messages $M \in \{0,1\}^*$, we have $\mathrm{Dec}_\mathrm{K}(\mathrm{Enc}_\mathrm{K}(M)) = M$.

B.1.2 Security Model

For symmetric encryption, we use the same security notion as in [33], i.e., ciphertext indistinguishability. Informally, ciphertext indistinguishability ensures that an attacker cannot distinguish between encryptions of any two messages. More formally, it is defined as follows. Let λ be a security parameter and let $\mathrm{SE} = (\mathrm{Enc}, \mathrm{Dec})$ be a symmetric encryption scheme. Consider the following game between a challenger and attacker. The challenger first picks a key $\mathrm{K} \in \{0,1\}^\lambda$. Then, the attacker specifies two messages M_0, M_1 and gives these to the challenger, who flips a coin $\beta \in_R \{0,1\}$ and returns $\mathrm{CT}_{\mathrm{sym}} \leftarrow \mathrm{Enc}_\mathrm{K}(M_\beta)$ to the attacker. The attacker outputs a guess β' for β. Then, $\mathrm{SE} = (\mathrm{Enc}, \mathrm{Dec})$ has indistinguishable ciphertexts if for all polynomial-time attackers in the game above holds that the advantage $\left| \Pr[\beta' = \beta] - \frac{1}{2} \right|$ is negligible

B.2 MAC Function

B.2.1 Formal Definition

We formally define a MAC function as follows. Let λ be the security parameter. A message authentication code (MAC) $(\mathrm{MAC}_{\mathrm{K_{MAC}}}, \mathrm{Vrfy}_{\mathrm{K_{MAC}}})$, where $\mathrm{K_{MAC}} \in \{0,1\}^\lambda$ is the MAC key, is defined by

- $\mathrm{MAC}_{\mathrm{K_{MAC}}}(M)$: On input message $M \in \{0,1\}^*$, this algorithm outputs a tag T.
- $\mathrm{Vrfy}_{\mathrm{K_{MAC}}}(M, \mathrm{T})$: On input message M and tag T, the algorithm returns 0 ("reject") or 1 ("accept").

The MAC is correct if for all keys $\mathrm{K_{MAC}} \in \{0,1\}^\lambda$ and all messages $M \in \{0,1\}^*$ it holds that if $\mathrm{T} \leftarrow \mathrm{MAC}_{\mathrm{K_{MAC}}}(M)$, then $\mathrm{Vrfy}_{\mathrm{K_{MAC}}}(M, \mathrm{T}) = 1$.

B.2.2 Security Model

For MACs, we use the notion of security against one-time chosen-message attacks. Let λ be the security parameter, and let $(\mathrm{MAC}_{\mathrm{K_{MAC}}}, \mathrm{Vrfy}_{\mathrm{K_{MAC}}})$ be a message authentication code. Consider the following game between challenger and attacker. The challenger first picks a key $\mathrm{K_{MAC}} \in \{0,1\}^{\lambda}$. The attacker sends a message M to the challenger, who returns a tag $\mathrm{T} \leftarrow \mathrm{MAC}_{\mathrm{K_{MAC}}}(M)$. Then, the attacker outputs a pair (M', T'). The attacker succeeds if $(M, \mathrm{T}) \neq (M', \mathrm{T}')$ and $\mathrm{Vrfy}(M', \mathrm{T}') = 1$.

B.3 Special Commitment Scheme

B.3.1 Formal Definition

The special commitment scheme that we use is defined as follows. Let λ be the security parameter.

- $\mathrm{ESetup}(\lambda) \rightarrow \mathrm{pub}$: Define hashes $h_1 \colon \{0,1\}^{448} \rightarrow \mathbb{Z}_p$ and $h_2 \colon \{0,1\}^{448} \rightarrow \{0,1\}^{\lambda}$, and set $\mathrm{pub} = (h_1, h_2)$.
- $\mathrm{ES}(\lambda, \mathrm{pub}) \rightarrow (\mathrm{rand}, \mathrm{com}, \mathrm{dec})$: Generate $\mathrm{dec} \in_R \{0,1\}^{448}$, and compute $\mathrm{com} = h_1(\mathrm{dec})$ and $\mathrm{rand} = h_2(\mathrm{dec})$.
- $\mathrm{ER}(\mathrm{pub}, \mathrm{com}, \mathrm{dec}) \rightarrow \mathrm{rand}$: Generate $\mathrm{rand} \leftarrow h_2(\mathrm{dec})$.

The special commitment scheme is correct if for all $(\mathrm{rand}, \mathrm{com}, \mathrm{dec}) \leftarrow \mathrm{ES}(\lambda, \mathrm{pub})$ holds that $\mathrm{ER}(\mathrm{pub}, \mathrm{com}, \mathrm{dec}) = \mathrm{rand}$.

B.3.2 Security Model

A special commitment scheme $(\mathrm{ESetup}, \mathrm{ES}, \mathrm{ER})$ is secure if it is hiding and binding.

- **Hiding:** Consider an attacker and a challenger. Then, the challenger runs $\mathrm{pub} \leftarrow \mathrm{ESetup}(\lambda)$ and flips a coin $\beta \in_R \{0,1\}$. If $\beta = 0$, then $\mathcal{C}$ generates a random $\mathrm{rand} \in_R \{0,1\}^{\lambda}$, and otherwise, it runs $(\mathrm{rand}, \mathrm{com}, \mathrm{dec}) \leftarrow \mathrm{ES}(\lambda, \mathrm{pub})$. It shares $(\lambda, \mathrm{pub}, \mathrm{rand}, \mathrm{com})$ with the attacker, who then outputs a guess β' for β. The scheme is hiding if for all such attackers, it holds that the advantage $\left|\Pr[\beta = \beta'] - \frac{1}{2}\right|$ is negligible.
- **Binding:** Consider an attacker and a challenger. Then, the challenger runs $\mathrm{pub} \leftarrow \mathrm{ESetup}(\lambda)$, and shares $(\mathrm{rand}, \mathrm{com}, \mathrm{dec}) \leftarrow \mathrm{ES}(\lambda, \mathrm{pub})$ with the attacker. Then, it is computationally infeasible for the attacker to find $\mathrm{dec}' \neq \mathrm{dec}$ such that $\mathrm{ER}(\mathrm{pub}, \mathrm{com}, \mathrm{dec}') = \mathrm{rand}$, i.e., for output $\mathrm{dec}' \neq \mathrm{dec}$ of the attacker, it holds that the success probability $\Pr[\mathrm{ER}(\mathrm{pub}, \mathrm{com}, \mathrm{dec}') = \mathrm{rand}]$ is negligible. The scheme is binding if this holds for all such attackers.

C Description of Our CCA-Secure Scheme

We give a simplified description (using the simplified description of TKN20 in Section 3.2.10) of our CCA-secure scheme below.

- $\mathrm{Setup}(\lambda) \rightarrow (\mathrm{MPK}, \mathrm{MSK})$: The setup outputs the master public-secret key pair (MPK, MSK), where $H_i \colon \{0,1\}^* \rightarrow \mathbb{G}_1$ with $i \in \{0,1\}$ are two hash functions (modeled as random oracles), $\mathrm{KDF} : \mathbb{G}_T \rightarrow \{0,1\}^{\lambda}$ is a key derivation function [17], $\mathrm{SE} = (\mathrm{Enc}, \mathrm{Dec})$ is a symmetric encryption scheme, pub are the public parameters generated with the setup ESetup of a special commitment scheme, $\mathrm{MSK} = (\alpha, b)$, and

$$\mathrm{MPK} = (\mathrm{SE}, \mathrm{pub}, p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, H_0, H_1, A = e(g_1, g_2)^{\alpha}, B = g_1^b).$$

- $\mathrm{KeyGen}(\mathrm{MSK}, (S, \psi_{\mathrm{lab}})) \rightarrow \mathrm{SK}_S$: On input a set of attribute values S and the associated labeling map $\psi_{\mathrm{lab}} \colon S \rightarrow \{0,1\}^*$, which maps the attributes in the set S to labels (represented as strings), it outputs the secret key SK_S as

$$\mathrm{SK}_S = (S, K_1 = g_1^{\alpha + rb}, K_2 = g_2^r,$$
$$\{K_{3,\mathrm{att}} = (H_0(\psi_{\mathrm{lab}}(\mathrm{att})) \cdot H_1(\psi_{\mathrm{lab}}(\mathrm{att}))^{x_{\mathrm{att}}})^r\}_{\mathrm{att} \in S},$$
$$K_{3,\mathrm{CCA}} = H_0(\mathrm{CCA})^r, K_{4,\mathrm{CCA}} = H_1(\mathrm{CCA})^r),$$

  where $r \in_R \mathbb{Z}_p$ is a randomly generated element in $\mathbb{Z}_p$ and x_{att} denotes the representation of att in $\mathbb{Z}_p$.

- $\mathrm{Encrypt}(\mathrm{MPK}, \mathbb{A}, M) \rightarrow \mathrm{CT}_{\mathbb{A}}$: On input a plaintext message $M \in \{0,1\}^*$ and an access policy $\mathbb{A} = (\mathbf{A}, \rho, \rho_{\mathrm{lab}}, \overline{\rho}, \tau)$—where $\tau \colon \{1, \ldots, n_1\} \rightarrow \{1, \ldots, m\}$ is a function that maps each row that is associated with the same label to a different integer in $\{1, \ldots, m\}$, with m being the maximum number of times that a label occurs in the policy—it first extends the policy $\mathbb{A}$ to $\mathbb{A}'$ such that it applies an AND operator to $\mathbb{A}$ and the attribute label-value pair `CCA: com` (where com is defined as below), and outputs a ciphertext $\mathrm{CT}'_{\mathbb{A}'}$ as

$$\mathrm{CT}'_{\mathbb{A}'} = (\mathbb{A}, \mathrm{com} \leftarrow h_2(\mathrm{dec}), C \leftarrow \mathrm{Enc}_{\mathrm{K}}(\mathrm{dec} \| M), \mathrm{CT}_{\mathbb{A}'},$$
$$\mathrm{T} = \mathrm{MAC}_{\mathrm{K}'}(\mathbb{A} \| \mathrm{com} \| C \| \mathrm{CT}_{\mathbb{A}'})),$$

  so that

$$\mathrm{CT}_{\mathbb{A}} = (C_1 = g_2^s, \{C_{2,l} = g_2^{s_l}\}_{l \in \{1, \ldots, m\}},$$
$$\left\{C_{3,j} = B^{\lambda_j} \cdot (H_0(\rho_{\mathrm{lab}}(j)) \cdot H_1(\rho_{\mathrm{lab}}(j))^{x_{\rho(j)}})^{s_{\tau(j)}}\right\}_{j \in \chi_0},$$
$$\left\{C_{3,j} = B^{-\lambda_j} \cdot H_0(\rho_{\mathrm{lab}}(j))^{s_{\tau(j)}},\right.$$
$$\left. C_{4,j} = B^{x_{\rho(j)} \lambda_j} \cdot H_1(\rho_{\mathrm{lab}}(j))^{s_{\tau(j)}}\right\}_{j \in \chi_1}),$$

  where $s, s_1, \ldots, s_m, v_2, \ldots, v_{n_2+1} \in_R \mathbb{Z}_p$ are randomly generated elements in $\mathbb{Z}_p$, $\lambda_j = A_{j,1} s + \sum_{k \in \{2, \ldots, n_2+1\}} A_{j,k} v_k$, $\chi_i = \{j \in \{1, \ldots, n_1 + 1\} \mid \overline{\rho}(j) = i\}$ for $i \in \{0,1\}$, $\mathrm{K} \leftarrow \mathrm{KDF}(A^s)$, $\mathrm{dec} \in_R \{0,1\}^{448}$ and $\mathrm{K}' \leftarrow h_1(\mathrm{dec})$.

- Decrypt($\mathrm{SK}_S, \mathrm{CT}_{\mathbb{A}'}$) $\rightarrow M'$: On input the ciphertext $\mathrm{CT}_{\mathbb{A}'}$ (where $\mathbb{A}'$ is an AND-composition of policy $\mathbb{A}$ and `CCA: com`), and a secret key SK_S, it first checks whether S satisfies the $\mathbb{A}$. If not, then it aborts. Otherwise, it first determines $\Upsilon_0 = \{j \in \chi_0 \mid \rho(j) \in S\}$, $\Upsilon_1 = \{j \in \chi_1 \mid \rho(j) \notin S \wedge \rho_{\mathrm{lab}}(j) \in \psi_{\mathrm{lab}}(S)\}$, $\Upsilon = \Upsilon_0 \cup \Upsilon_1$ and $\{\varepsilon_j\}_{j\in\Upsilon}$ such that $\sum_{j\in\Upsilon} \varepsilon_j \mathbf{A}_j = (1, 0, ..., 0)$, then computes $e(g_1, g_2)^{\alpha s}$ as in the decryption of TKN20 (Section 3.2.10), where a key can be generated for `CCA : com` by computing $K_{3,\mathrm{CCA}} \cdot K_{4,\mathrm{CCA}}^{x_{\mathrm{com}}}$, then retrieves:

$$\begin{aligned} \mathrm{K} &\leftarrow \mathrm{KDF}(e(g_1,g_2)^{\alpha s}) \\ \mathrm{dec} \| M &\leftarrow \mathrm{Dec}_{\mathrm{K}}(C) \\ \mathrm{K}' &\leftarrow h_1(\mathrm{dec}), \end{aligned}$$

and verifies:

$$\begin{aligned} h_2(\mathrm{dec}) &\stackrel{?}{=} \mathrm{com} \\ \mathrm{Vrfy}(\mathbb{A} \| \mathrm{com} \| C \| \mathrm{CT}_{\mathbb{A}'}, \mathrm{T}) &\stackrel{?}{=} 1. \end{aligned}$$

If both checks pass, then the decryption returns M, and if not, it returns an error message.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

GLogS: Interactive Graph Pattern Matching Query At Large Scale

Longbin Lai[1]*, *Yufan Yang*[2]*, *Zhibin Wang*[3], *Yuxuan Liu*[2], *Haotian Ma*[2], *Sijie Shen*[1], *Bingqing Lyu*[1], *Xiaoli Zhou*[1], *Wenyuan Yu*[1], *Zhengping Qian*[1], *Chen Tian*[3], *Sheng Zhong*[3], *Yeh-Ching Chung*[2] *and Jingren Zhou*[1]

[1]Alibaba Group, China
[2]The Chinese University of Hong Kong, Shenzhen
[3]Nanjing University

Abstract

Interactive GPM (iGPM) is becoming increasingly important, where a series of graph pattern matching (GPM) queries are created and submitted in an interactive manner based on the insights provided by the prior queries. To solve the iGPM problem, three key considerations must be taken into account: performance, usability and scalability, namely if results can be returned in a timely manner, if queries can be written in a declarative way without the need of imperative fine-tune, and if it can work on large graphs. In this paper, we propose the GLogS system that allows users to interactively submit queries using a declarative language. The system will compile and compute optimal execution plans for the queries, and execute them on an existing distributed dataflow engine. In the evaluation, we compare GLogS with the alternatives systems Neo4j and TigerGraph. GLogS outperforms Neo4j by 51× on a single machine due to better execution plans. Additionally, GLogS can scale to processing large graphs with distributed capability. While compared to TigerGraph, GLogS is superior in usability, featuring an optimizer that can automatically compute optimal execution plans, eliminating the need of manual query tuning as required in TigerGraph.

1 Introduction

Graph pattern matching (GPM) aims to compute the mappings in a data graph that match a given small pattern graph, and it plays an important role in a variety of applications covering bioinformatics [3, 24, 37], chemistry [17, 23], social/web network analysis [20, 26], and recently in enhancing the expressive power of graph neural network [15, 53, 55].

Increasingly, interactive GPM (iGPM) is becoming critical for data scientists to mine relationships, identify frauds or detect intrusions from a variety of large graphs in real life. In these scenarios, data scientists create and submit a series of GPM queries in an interactive manner based on the insights provided by the results of prior queries. For example, we demonstrate a simplified application scenario as follows.

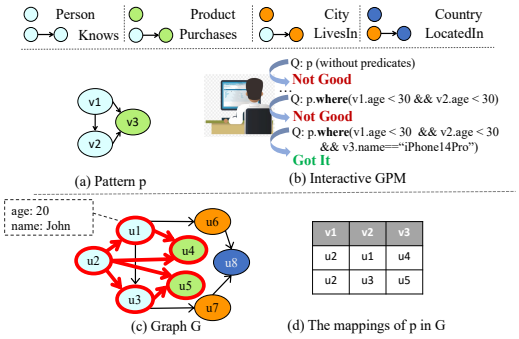


Figure 1: Example of interactive graph pattern matching, where a user will interactively submit graph pattern matching queries (see Example 2.1) to explore the graph.

Example 1.1. In Figure 1, a user is exploring recommendation rules in an e-commerce graph, which maintains relationships such as "Purchases" between persons and products. The user is specifically looking at a pattern (Figure 1(a)) of co-purchasing among pairs of people who are acquainted. If such co-purchasing occurs very frequently among these pairs in historical data, a recommendation rule may be created to recommend the product that has been purchased by one person to his/her friends who have not yet purchased it. The user tries different patterns and constraints (using predicates) through a series of interactive queries. In Figure 1(b), the user decides to create a rule suggesting that young people ("*age* < 30") tend to co-purchasing the "iPhone 14 Pro", rather than an arbitrary product.

In the above scenario of iGPM, it is necessary to consider the requirements of *performance*, *scalability* and *usability* simultaneously. Performance allows users to quickly obtain useful insights from the "trial-and-error"

*Equal Contribution.

process. Usability enables users to easily present arbitrary GPM queries. Scalability is also crucial as it is now common to handle large-scale graphs. Due to the computation-intensiveness of GPM queries [14, 29], it is already difficult for a graph expert to tune the execution [13, 25, 54]. The problem becomes even more complex in iGPM, where users may not be experts and the queries can involve intricate patterns and optional predicates. Therefore, the following features are essential to meet the above requirements.

Declarative Language. A declarative query language can provide users convenience and flexibility to express complex GPM queries.

Automatic Optimization. Automatic optimization allows non-expert users to focus on exploring valuable patterns without having to worry about the challenging task of performance tuning.

Distributed Execution. With the graph partitioned across the cluster, distributed execution is expected to spread the workloads accordingly.

However, the systems that are potentially usable for iGPM, including Neo4j [33] and TigerGraph [18], all fall short in providing one or more above features (Section 2). In this paper, we propose the GLogS (name after GLogue, see Section 5.3) system to fill in the gap. Our goal is to give the users the convenience and flexibility of presenting GPM queries interactively, and to have the system deal with the complexities that arise from compilation, query optimization, and distributed execution. We mainly make the following technical contributions.

(1) A compilation stack that compiles declarative GPM queries into distributed programs. We adopt Gremlin's declarative `match` step, an industrial-strength query language, for expressing GPM queries. The `match` step, after compilation and optimization, will be transformed into a program that can be executed on a distributed dataflow engine.

(2) An optimizer that can automatically derive optimal execution plans for GPM queries. While analyzing the execution plans of Neo4j, we have identified two critical impact factors of deriving good execution plans for GPM: worst-case optimal execution plan [4] and high-order statistics [12]. We take into considerations both factors to design and implement the optimizer for GLogS.

(3) A system that allows users to interactively submit and efficiently execute GPM queries at large scale. We build the GLogS system upon the existing distributed dataflow engine GAIA [38] to leverage its optimization for graph queries. In the evaluation based on the LDBC benchmark, we compare GLogS with Neo4j and TigerGraph. GLogS outperforms Neo4j by 51× on a single machine due to better execution plans. Additionally, GLogS can scale to handle large graphs with distributed capability. While compared to TigerGraph, GLogS is superior in usability, featuring an optimizer that can automatically compute optimal execution plans, eliminating the need of manual query tuning as required in TigerGraph.

2 Background and Challenges

2.1 The Problem of iGPM

We adopt the property graph model [6] for usability. A property graph $G(V_G, E_G)$ is a directed labelled graph, as shown in Figure 1, in which each vertex $u \in V_G$ models an entity, and each edge $(u_s, u_t) \in E_G$ models the relationship from a source vertex u_s to a target vertex u_t. We call the edge (u_s, u_t) the adjacent edge of u_s and u_t, and u_s (resp. u_t) is an in neighbor (resp. out neighbor) of u_t (resp. u_s). A vertex u (an edge is analogously defined) is assigned a globally unique identifier (Id) and a label (Label) to indicate is type. Moreover, it can carry a collection of key-value pairs as the properties. We use u.key to denote accessing u's property of given key.

A pattern $p(V_p, E_p)$ is a small *connected* graph. Given a pattern p and graph G, the graph pattern matching (GPM) problem aims to compute all mappings $Q_G(p)$ of the pattern in the graph G, where each mapping $f \in Q_G(p)$ matches the pattern vertices[1] to a set of non-duplicate graph vertices one by one, so that if there is a pattern edge between two pattern vertices, there must be a graph edge between the two matched graph vertices. For a pattern vertex v, we use $f(v) = u$ to obtain the matched graph vertex u. The number of mappings is called the frequency of the pattern in the graph, denoted as $\mathscr{F}_G(p)$. When the context is clear, the subscript of G in above notations may be omitted (i.e. $Q(p)$). Predicates can be specified, while we mostly omit predicates to focus on the pattern in the paper. Details of how we handle predicates are in our open-source page [46].

Example 2.1. In Figure 1, there are two mappings of the triangle pattern p in the graph as shown, and thus the frequency of p is 2. Specifically, a mapping f matches v_1 to u_2 and v_2 to u_1, namely $f(v_1) = u_2$ and $f(v_2) = u_1$. Obviously, there is a graph edge (u_2, u_1) corresponding to the pattern edge (v_1, v_2). The predicate "$v_1.age < 30$ && $v_2.age < 30$" in Figure 1(b), constrains that the vertices matching v_1 and v_2 must have "age" smaller than 30.

We target the iGPM scenario to process GPM queries on large-scale property graphs in an interactive context.

2.2 Solving iGPM using Existing Systems

Graph databases [2, 8, 18, 27, 33] allow users to interactively query the graph using declarative query languages,

[1] The details of matching edges are not discussed as they are similar to matching vertices.

and thus have the most potential to be deployed for iGPM. However, they often lack support for automatic optimization or distributed execution, and thus cannot meet performance, usability and scalability at the same time. We discuss Neo4j and TigerGraph as representatives. Other related systems are surveyed in Section 8. Neo4j [33] is one of the most popular graph databases, but is limited by its single-machine design and insufficient optimizer, leading to poor scalability and performance as reported in previous studies [38, 44, 48]. TigerGraph [18], on the other hand, is a distributed system that can scale well but lacks an automatic optimizer, requiring users to manually tune the plan for good performance, which significantly limits its usability. In addition, users need to pre-install the queries before they can be executed on TigerGraph. The pre-installation involves native code generation and compilation, which can take 1 to 3 minutes per query in our evaluation and may not be tolerable in the interactive context.

2.3 Challenges of Solving iGPM

It's challenging to develop an iGPM system with performance, usability and scalability. Here, we discuss the issues arise from query compilation and optimization.

Compilation. Compiling a declarative query language itself is non-trivial, and the interactive context introduces extra difficulties. Due to the timeliness of iGPM, it's infeasible to generate native codes from the queries and process a time-consuming online compilation like TigerGraph [18]. Preparing store procedurals for a set of queries offline is not possible, as the useful patterns remain unknown in advance.

Optimization. The automatic optimization of GPM lies at the core of an iGPM system, but it's difficult to design such an optimizer for real-life queries. To see this, in Figure 2, we've demonstrated the execution plans Plan_G and Plan_N of a benchmark query derived by the optimizers of GLogS and Neo4j, respectively. For now, one only needs to know that a better execution plan typically, if not definitely, produces less intermediate results, which are a collective of the mappings of all intermediate patterns that must be computed during the execution. We mark in Figure 2 the corresponding intermediate pattern frequencies in the benchmark graph G_1 (Table 1). Obviously, Plan_N produces much larger intermediate results than Plan_G. We execute the two plans in our system, and Plan_N not only runs orders of magnitude slower, but also consumes significantly larger memory, than Plan_G. This demonstrates that Neo4j's optimizer is insufficient to handle such complex GPM queries.

There are two main reasons for this. Firstly, the execution plan given by Neo4j cannot guarantee *worst-case optimality*. Secondly, it uses only *low-order statistics* to estimate the cost of a plan, which can cause the resulting plan to have small estimated cost even though it actually costs heavily. We elaborate in details.

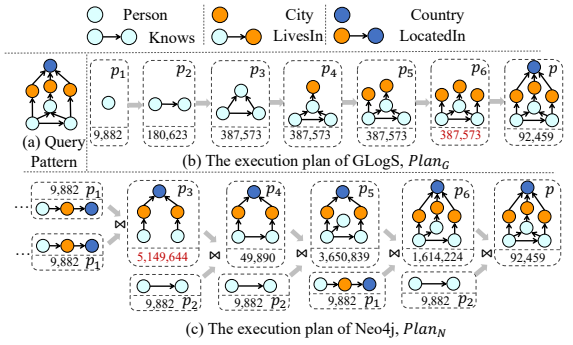


Figure 2: Execution plans of GPM. Certain trivial steps in the plans are not shown for clarity.

Worst-case optimality. An execution plan for computing a pattern p is worst-case optimal, if the frequency of any *intermediate pattern* in the plan does not exceed $\mathscr{F}(p)$ in the worst case. Here, an intermediate pattern in an execution plan refers to a subgraph (or sub-pattern, interchangeably) of the queried pattern whose mappings must be computed during the execution. For instance, the patterns p_1-p_6 in Plan_G (all plans in this section are referred to Figure 2) are intermediate patterns of p. The process of solving GPM typically involves operations of binary join and vertex expansion. Briefly, binary join involves performing a hash join on the mappings of two input patterns in order to produce the results of the output pattern. For example, in Plan_N, p_2 is joined with p_3 to produce p_4. Execution plans that rely *solely* on binary joins, such as Plan_N, are called binary-join plans. However, these plans may not guarantee worst-case optimality [29]. Neo4j's plan actually falls into this category, which is a dominant factor of its poor performance [48].

Alternatively, Ammar et al.[4] have looked into Ngo's algorithm[35] for GPM optimization. The key operation to this algorithm is vertex expansion, which involves expanding a base pattern by adding one more vertex to it. In Plan_G, for instance, p_2 is expanded to p_3 in this way. Begin with a base pattern that is a vertex, the algorithm processes vertex expansions iteratively until the desired pattern is obtained. According to Ngo's algorithm, we can obtain an execution plan that is worst-case optimal, such as Plan_G for GPM.

Recent research [1, 32, 54] has shown that the best possible execution plans for GPM must incorporate both binary joins and vertex expansions. Consequently, our optimizer must be capable of handling this hybrid strategies. It's crucial to note that to achieve worst-case optimality, such hybrid plans must carefully consider the use of binary joins, as will be explained in Section 5.4.

High-order statistics. An optimal execution plan for a GPM query is the plan that has the smallest cost. The pattern frequencies are essential for evaluating the cost of an execution plan. In large-scale graphs, it is more feasible to estimate these frequencies, rather than trying to exactly compute them. Neo4j uses Low-order statistics such as the number of vertices and edges (of each type) to estimate pattern frequencies by assuming independent existence of graph edges [34]. However, this assumption is too idealistic and can lead to inaccurate cost estimation and poor execution plans in practice. To address this issue, we follow Mhedhbi et al. [32] to exploit the *high-order statistics* [12] of the graph.

Definition 2.1. The high-order statistics of a graph refer to the frequencies of a series of small patterns (also known as motifs [3]), from the smallest single-vertex patterns to the largest patterns that are *complete* graphs of k vertices. Here, k is called the high-order *level*, and must be at least 3 to avoid degrading to low-order.

Mhedhbi et al. [32] and us have both demonstrated the effectiveness of the high-order statistics. However, their computation is at least as costly as the widely recognized computation-intensive workload of graph pattern mining [45]. To reduce the computation cost, Mhedhbi et al. [32] have proposed using a sampling technique that matches a randomly selected portion of data vertices and edges at runtime. Nevertheless, the sampling technique is difficult to apply to large-scale graphs that are partitioned in the cluster.

3 System Overview

We've built the GLogS system for iGPM, as shown in Figure 3, to address the challenges in Section 2.3. The system allows users to interactively submit their GPM queries using Gremlin's declarative `match()` step [7]. An example of the Gremlin code for the triangle pattern is presented. Because GPM is computationally intractable [28], users can optionally specify a timeout to prevent the query from running for an unreasonable amount of time. It is worth noting that relational operations, including projection, ordering and grouping, may also be applied in iGPM. However, for queries that involve these operations, GLogS processes them only after the execution of GPM. Therefore, their computation complexity is dominated by that of GPM, and will not be further discussed in this paper.

Frontend Module. The frontend machine runs the processors of a *pattern parser*, a *plan optimizer* and a GLogue *manager* for parsing and optimizing the GPM queries. The pattern parser parses a Gremlin query into a *language-agnostic* structure called `PatternDesc`. The

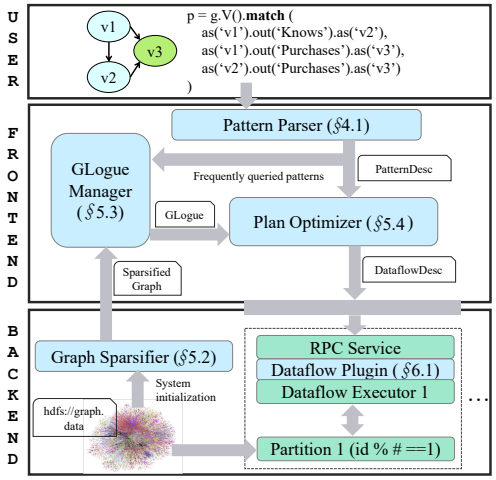


Figure 3: System overview. Blue components are the focus of this paper, with the paper's sections indicated.

key objective of designing `PatternDesc` is to decouple the query language and the optimizer. This enables easy integration with other query languages such as Neo4j's Cypher [16]. For given `PatternDesc`, the plan optimizer aims to produce an execution plan that has the smallest-possible cost based on a generic cost model. The cost model considers the high-order statistics of the graph, which are maintained in a novel graph-based structure called GLogue. The GLogue manager handles the construction of the GLogue by computing the frequencies on the sparsified graph for patterns up to k vertices (i.e. with high-order level k) when the system is initiated. It also buffers the frequently queried patterns whose frequencies are missing from the GLogue, and launches a procedural periodically to append these frequencies to the GLogue.

Backend Module. The backend module consists of a *distributed dataflow engine*, a *graph store* and a *graph sparsifier*, spreading across a cluster of n computing nodes. We have built the GLogS system on an existing dataflow engine, which organizes n executors, each corresponding to a computing node in the cluster. A declarative `DataflowDesc` is constructed from the execution plan produced by the plan optimizer, which embeds the computing instructions of GPM in a dataflow that is a directed acyclic diagram (DAG). The `DataflowDesc` is then distributed to all executors via RPC services to launch the computation in parallel. To make the `DataflowDesc` executable, a library of dataflow plugin is implemented that contains the generated code and a job assembler to assemble the distributed program. The dataflow plugin is required to co-compile offline with the underlying dataflow engine, which bypasses a costly online native-code compilation as TigerGraph [18].

The graph store manages the partitioned graph data in the cluster. As it is not the main focus of this paper, for

simplicity, we adopt in-memory and immutable graph store, where the graph data is partitioned using a hash partition strategy, namely the vertex u will be placed on the partition of "u.Id % # partitions", together with all its properties and adjacent (both in and out) edges. Such a simple yet widely used partition strategy[4, 29, 38] may lead to load skew, which can potentially impact query performance. Nonetheless, a well-optimized execution plan is still the key to the efficiency and scalability of GPM. Therefore, rather than exploring alternative partition strategies [52], we employ the simple strategy and focus on query optimization in this work. The i[-th] partition of the graph is co-located with the i[-th] executor of the dataflow engine. Moreover, The raw graph data are pre-partitioned, encoded and stored in a distributed file system such as *HDFS*. Each dataflow executor loads its partition into the main memory while starting up the system. During system initialization, a graph sparsifier will be simultaneously launched as loading graph data, which is responsible for sparsifying the large-scale graph into a small graph that can fit into the main memory of the frontend machine. The sparsified graph will be serialized to a persistent store to prevent the need for re-sparsification when the system is restarted.

4 Compiling Declarative GPM Queries

We demonstrate in Figure 4 the process of compiling the declarative Gremlin's `match()` step for a GPM query into distributed dataflow program.

4.1 The Pattern Parser

As shown in Figure 4(a), in Gremlin's `match()` step, a pattern is described as a collection of clauses in the form of "`as().[in|out]().as()`", in which the start and end `as()` steps identify the two vertices with tags that are unique in the pattern, and the `in()` or `out()` step in between expresses the edge that connects the two vertices. In this simplest[2] form of a `match()`, each clause expresses an edge in the pattern. Given a Gremlin's `match()` step, the pattern parser utilizes the ANTLR tool [21] (officially provided by Gremlin) to parse a query into an Abstract Syntax Tree, from which a `PatternDesc` is built, as illustrated in Figure 4(b).

We first define two *computing primitives* called `GetV` and `GetE` for describing GPM queries. A `GetV` primitive is a 4-tuple (eTag, tag, label, [Source|Target]) that encodes the semantics of matching the vertex with "tag" as the source or target vertex of an "eTag" edge. A `GetE`(vTag, tag, label, [In|Out]) encodes matching the edge with "tag" as the in or out edge of a "vTag" vertex.

A sentence that is an ordered sequence of `GetV` and `GetE` is then used to encode the semantics of a clause in `match()`, and a `PatternDesc` is composed of a collection of sentences. The semantics are self-explanatory, and we just discuss some special use cases in the first sentence of Figure 4(b). Observe that the first `GetV` has the "eTag" field unspecified (`NA`), which means that the vertex may not be bound to any prior edge and should match all vertices in the graph. In the `GetE`, the "tag" field is specified as an empty `String`. This tells the runtime that the matched edge (also applied for `GetV`) should not be kept in the results, which is useful when only a part of the matched instances are needed in practice. Following the `GetE`, a `GetV` has an empty "eTag", which means the vertex must be obtained directly from this "previous" edge.

Observe that we include the label information in `GetV` that is not actually given in the Gremlin query. In GLogS, while loading the graph data, we can meanwhile extract the *meta connections* that maintain the possible types of source and target vertices of each edge type. For example, a Purchases edge can only connect a Person to a Product. Such meta connections not only help us validate user queries, but also reduce the number of patterns stored in the GLogue (Section 5.3).

4.2 The Dataflow Embedding

In the plan optimizer (Section 5.4), an optimized execution plan will be computed for the `PatternDesc`, which will be further embedded into a `DataflowDesc` that describes how to compute the pattern in a dataflow engine.

In a dataflow engine, a dataflow is a directed acyclic graph (DAG) that abstracts the computation, in which a vertex stands for an operator that defines the computing logic, and an edge between two operators o_1 and o_2 represents the data channel such that the output of o_1 is the input of o_2. In the task of GPM, the input and output data of each operator in a dataflow are mappings of the patterns. We introduce five operators in this paper:

- `Source(udf)`: A `Source` operator specifies the input data of the dataflow program, which are a collection of vertices in the graph.
- `Sink(udf)`: The `Sink` operator (only one allowed) writes the results to the output channel (e.g. an RPC port) that users can access.
- `Map(udf)`: For each input item, a `Map` operator computes *exactly* one data item using the given user-defined function (udf).
- `FlatMap(udf)`: For each input item, a `FlatMap` operator can produce *arbitrary* (none, single or multiple) number of data items via the udf.
- `Join(key1, key2, udf)`: A `Join` operator consumes two input data, extracts the corresponding

[2] The other more complex forms only bring in engineering details.

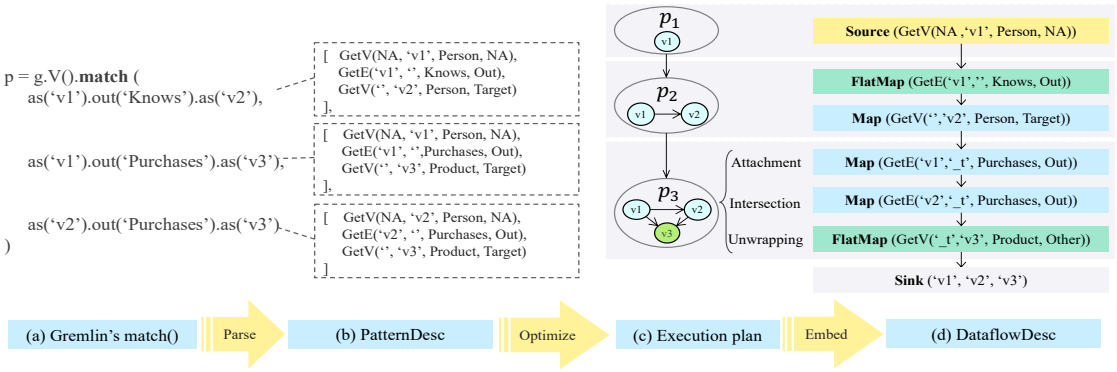


Figure 4: The process of compiling a Gremlin's `match` step into a `DataflowDesc`.

keys, and conducts a join via the `joinFunc` on the two input data.

In a nutshell, a `DataflowDesc` for GPM is a dataflow that embeds the GPM primitives of such as `GetV` and `GetE` in the above operators as the udfs. Figure 4(d) illustrates a `DataflowDesc` for computing the triangle pattern. We look into the first three operators for now, which describe the computation of the mappings for v_1 and v_2. First, v_1 is matched in a `Source` operator that consumes all Person vertices from the graph. Then for each vertex that matches v_1, a `FlatMap` operator is assigned to traverse its out edges, which is reasonable as a vertex typically has more than one adjacent edges in the graph. The last `Map` operator extracts the target vertex from each edge to match v_2. More details of how we compute the given `DataflowDesc` will be discussed in the next section.

5 Automatic Optimization

This section covers the automatic optimization of GPM queries, which is a collaboration of the system components of graph sparsifer, GLogue manager, and plan optimizer. For a graph vertex u, we denote $\mathsf{Nbr}[elabel](u)$ as the neighbors[3] of u in the graph G subject to the edge label constraint. Given two graphs G_1 and G_2, G_2 is a subgraph of G_1, denoted as $G_2 \subseteq G_1$, if $V_{G_2} \subseteq V_{G_1}$ and $E_{G_2} \subseteq E_{G_1}$. Furthermore, G_2 is an induced subgraph of G_1, if it contains *all* edges in G_1 among V_{G_2}. For two sets S_1 and S_2, we denote $S_1 \setminus S_2$ as the set of elements in S_1 but not in S_2.

5.1 Execution Plan and Cost Model

We first introduce the execution plan for GPM and define its cost, which allows us to search for the optimal execution plan as the one with the smallest cost.

[3]Note that the edges can be in out, in or even both directions, but we omit the direction in the notation for simplicity.

Execution Plan. To allow the optimizer to derive hybrid execution plans as mentioned in Section 2.3, we consider two basic operations: the *binary join* and *vertex expansion* that are critical to fulfil the binary joins and worst-case optimal joins, respectively. A binary join, denoted as $\mathsf{Join}(\{p_{s_1}, p_{s_2}\} \to p_t)$, conducts hash join operation for $Q(p_{s_1})$ and $Q(p_{s_2})$ on the join key of $V_{p_{s_1}} \cap V_{p_{s_2}}$ to compute the results of $Q(p_t)$. The operation of vertex expansion needs further explanation.

Definition 5.1. Consider two patterns, p_s and p_t with $V_{p_t} \setminus V_{p_s} = \{v\}$, and $E_{p_t} \setminus E_{p_s} = \{e_1 = (v_1, v), e_2 = (v_2, v), \ldots, e_k = (v_k, v)\}$ without loss of generality. A vertex-expansion operation, denoted as $\mathsf{Expand}(p_s \to p_t)$, extends each mapping f of p_s by one more graph vertex corresponding to v. The newly matched graph vertex must be in the common neighbors of all $f(v_i)$ for $1 \le i \le k$, namely $\bigcap_{i=1}^{k} \mathsf{Nbr}[e_i.\mathsf{Label}](f(v_i))$.

Example 5.1. In Figure 4(c), it's clear that $V_{p_3} \setminus V_{p_2} = \{v_3\}$. This allows us to perform a vertex expansion $\mathsf{Expand}(p_2 \to p_3)$. We use the graph in Figure 1 to illustrate the process. For a given mapping $f = \{u_2, u_1\}$ of p_2, we can expand it to the mapping $\{u_2, u_1, u_4\}$ for p_3. Here, v_3 is matched to $\{u_4\}$, which is obtained by intersecting $\mathsf{Nbr}[\mathsf{Purchases}](u_2)$ and $\mathsf{Nbr}[\mathsf{Purchases}](u_1)$. We can similarly perform this process for the other mappings $\{u_1, u_3\}$ and $\{u_2, u_3\}$.

Given a queried pattern p, we denote an execution plan for computing p as $\mathsf{Plan}(p) = (\Phi = \{p_1, p_2, \ldots, p_n = p\}, \Gamma = [\tau_1, \tau_2, \ldots, \tau_m])$, where Φ represents a set of intermediate patterns and Γ is an ordered sequence of operations that can be either binary join or vertex expansion. For example, we have the execution plan in Figure 4(c) as $(\Phi = \{p_1, p_2, p_3\}, \Gamma = [\mathsf{Expand}(p_1 \to p_2), \mathsf{Expand}(p_2 \to p_3)])$.

Cost Model. With the execution plan $\mathsf{Plan}(p) = (\Phi, \Gamma)$, we propose the cost model as

$$\mathsf{Cost}(\mathsf{Plan}(p)) = \sum_{p' \in \Phi} \mathscr{F}(p') + \sum_{\tau \in \Gamma} \mathsf{Cost}(\tau). \quad (1)$$

The first part refers to the cost of accessing the intermediate results from the memory, which can be considered as the commmunication cost, namely, the cost of accessing remote memory. This is because the intermediate results are a collection of the output from all executors in the cluster, and the cost of accessing remote data is much greater than that of accessing local data. The second part stands for the cost of the operations, also known as the computation cost.

As any join algorithm must go through the data of both participants, the cost of a binary join is computed as

$$\mathsf{Cost}(\mathsf{Join}(\{p_{s_1}, p_{s_2}\} \to p_t)) = \alpha_j(\mathscr{F}(p_{s_1}) + \mathscr{F}(p_{s_2})), \tag{2}$$

where α_j is a normalized factor. We do not consider the joined results in Equation 2 because it must have been considered as the communication cost in Equation 1.

Consider a vertex expansion $\mathsf{Expand}(p_s \to p_t)$ in Definition 5.1, and let f be one mapping of p_s. The cost of the vertex expansion of f is dominated by intersecting the neighbors of $f(v_i)$ for $1 \le i \le k$, which has the complexity of $\sum_{i=1}^{k} |\mathsf{Nbr}[e_i.\mathsf{Label}](f(v_i))|$. Regarding e_i, let $\sigma_{e_i}(f) = |\mathsf{Nbr}[e_i.\mathsf{Label}](f(v_i))|$ be the vertex-expansion factor of the mapping f, and $\overline{\sigma_{e_i}}$ the average factor of all mappings. We have the cost of vertex expansion as

$$\begin{aligned}\mathsf{Cost}(\mathsf{Expand}(p_s \to p_t)) &= \alpha_{ve} \sum_{f \in Q(p_s)} \sum_{i=1}^{k} \sigma_{e_i}(f) \\ &= \alpha_{ve} \mathscr{F}(p_s) \sum_{i=1}^{k} \overline{\sigma_{e_i}},\end{aligned} \tag{3}$$

where α_{ve} is a normalized factor that, along with α_j in Equation 2, aligns the differences in computation cost of vertex expansion and binary join, as well as the communication cost and the computation cost.

5.2 Graph Sparsifier

While it is necessary to compute $\mathscr{F}(p)$ of any pattern p for Equation 1, it's cost-prohibitive to do so directly. The sampling technique proposed in [32] cannot be applied to large-scale graphs (see Section 2.3). Therefore, we explore the technique of graph sparsification [40, 49]. Specifically, during system initialization, the graph sparsifier will conduct sparsification on each partition of the graph to randomly preserve a subset of edges, and aggregate them at the frontend machine to form the sparsified graph G^*. It is obviously more feasible to compute the pattern frequencies on G^* than on the original graph G. Thus, we use $\mathscr{F}_{G^*}(p)$ (with normalization) as an estimation of $\mathscr{F}_G(p)$ for cost evaluation.

However, it's non-trivial to sparsify real-life graphs that can contain many different types of edges. A naive uniform sparsification [40, 49] adopts a uniform *sparsification ratio* (the probability of keeping an edge) for all edges during sparsification. Although such a naive approach can obtain unbiased estimation of $\mathscr{F}_G(p)$, but it still works poorly in our evaluation (Section 7). The main reason is that different types of edges can appear in rather skewed frequencies in real-life graphs. A less frequent type of edge, such as the LocatedIn edge in Figure 1 that appears in thousands compared to the Purchases edge in billions, is more likely to get eliminated during sparsification, causing the estimation to have large variance.

We also notice that there are sparsification [11, 42] and coarsening [31] algorithms based on spectral graph theory, aiming to offer a superior approximation via biased sampling. However, these algorithms emphasize preserving global statistics such as edge cut, rather than counting subgraphs that are local information. As a result, they may not be suitable for our task.

Regarding our task, we adopt the stratified sparsification [19] that treats each type of edges as an independent stratum, and assign each stratum an individual sparsification ratio. The stratified sparsification provides the flexibility in choosing the sparsification ratio, and we propose an optimization problem that aims to minimize the estimation variance through tuning the ratio. Before proposing our optimization problem, we first introduce the norm factors and discuss its unbiasedness. Let the sparsification ratios be $\Omega = \{\rho_1, \rho_2, \ldots, \rho_l\}$, where ρ_i denotes the ratio for the stratum of edges with label i without loss of generality. The following lemma holds.

Lemma 5.1. *Let* $\widetilde{\mathscr{F}(p)} = \prod_{e \in E_p} \frac{1}{\rho_{e.\mathsf{Label}}} \mathscr{F}_{G^*}(p)$. *We have* $\mathbb{E}[\widetilde{\mathscr{F}(p)}] = \mathscr{F}_G(p)$, *where* $\mathbb{E}[X]$ *denotes the expected value of a random variable* X.

Proof is in Section A.1.1. In the following, when we write $\mathscr{F}(p)$, we by default mean $\widetilde{\mathscr{F}(p)}$ if not otherwise specified.

The next question is how to specify the sparsification ratios. Given that the sparsified graph must reside in the frontend machine, we model an optimization problem subject to the memory constraint M, that minimizes the variance of the frequency estimation regarding a forged pattern p^* formed by all types of edges in the graph, as:

$$\begin{aligned}&\arg\min_{\Omega} \mathrm{Var}[\widetilde{\mathscr{F}(p^*)}] \\ &\text{s.t.} \sum_{i=1}^{l} s_i \rho_i \le M,\end{aligned} \tag{4}$$

where s_i denotes the frequency of the edges that have label i. The optimization problem achieves its optimum under the following condition:

$$\rho_i = \min(1, \frac{M}{l} \times \frac{1}{s_i}),$$

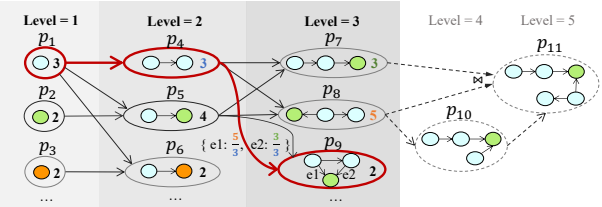


Figure 5: A fragment of GLogue. The value aside the pattern indicates its frequency in the graph in Figure 1(c). Note that the patterns in "Level=4/5" do not actually present in the GLogue.

Detailed derivation can be found in Section A.1.2. Interestingly, if $s_i \ll M$, ρ_i must be set to 1, which means that a very infrequent type of edge, such as the above LocatedIn edges, must not be ruled out during sparsification.

5.3 GLogue Manager

Our system aims to automatically derive the optimal execution plan for any arbitrary pattern. To do so, we follow the methodology presented in [32] to calculate the high-order statistics (Definition 2.1) of the graph. However, the approach in [32], which employs a table-based *catalogue* for retaining the high-order statistics, is not only expensive to construct but also challenging to apply when there are numerous complex pattern relationships in real-world graphs that need to be recorded, thereby making plan searching a difficult task. Instead, we have recognized the intrinsic suitability of graph structure for retaining complex relationships of this sort, and thus, we have proposed GLogue as a graph-based catalogue.

The GLogue is a hierarchical property graph as shown in Figure 5, in which each vertex is a pattern p at level $|V_p|$ with its frequency $\mathscr{F}(p)$ as the property. To ease lookup, the pattern will be encoded as a `String` using the technique of canonical labelling [10]. We say that the GLogue has k levels, if it maintains the high-order statistics up to level k. There're two types of edges in GLogue. The first type connects p_s and p_t in GLogue, if p_s can be expanded to p_t via vertex expansion. Regarding $\mathsf{Expand}(p_s \to p_t)$, the edge (p_s, p_t) records the vertex-expansion factors $\overline{\sigma_e}$ for all $e \in E_{p_t} \setminus E_{p_s}$. The second type corresponds to a binary join $\mathsf{Join}(\{p_{s_1}, p_{s_2}\} \to p_t)$, which introduces one edge from p_{s_1} to p_t with $(p_{s_2}, \mathscr{F}(p_{s_2}))$ as the property.

We deploy the GLogue manager in GLogS for the construction and maintenance of GLogue. When the system is initiated, with a threshold of level k, GLogue will be constructed from scratch to include all valid patterns with up to k vertices that satisfy the meta connections (see Section 4). While processing queries, the GLogue manager will buffer the incoming patterns (patterns only without predicates) from the users, and will launch a procedure to update the GLogue from the buffered patterns periodically.

Algorithm 1: The Plan Optimizer.

```
1  Function PlanOptimizer (GLogue, PatternDesc)
2      Construct a pattern p from the PatternDesc;
3      Let QSet organize all induced subgraphs of p by level;
4      Initialize a PlanMap to record {p : (plan, cost)} with
         patterns in level 1 and 2 pre-computed;
5      for 3 ≤ level ≤ |V_p| do
6          for p ∈ QSet[level] do
7              searchPlan (p, PlanMap, GLogue);
8      return PlanMap.get(p);
9  Function searchPlan (p, PlanMap, GLogue)
10     Initialize Plan(p) and Cost(Plan(p)) ← ∞;
11     for edge = (p_{s_1}, p) ∈ GLogue.getEdges(p) do
12         (plan1, cost1) ← PlanMap.get(p_{s_1});
13         if edge is a vertex extension then
14             Compute a new plan' by merging plan1 and
                 Expand(p_{s_1} → p);
15         else if edge{(p_{s_2}, F(p_{s_2}))} is binary join then
16             (plan2, cost2) ← PlanMap.get(p_{s_2});
17             Compute a new plan' by merging plan1, plan2
                 and Join({p_{s_1}, p_{s_2}} → p);
18         Compute a new cost' of plan' by Equation 1;
19         if cost' < Cost(Plan(p)) then
20             Update Plan(p) as plan' and the cost as cost';
21     PlanMap.insert(p, (Plan(p), Cost(Plan(p))));
```

5.4 Plan optimizer

Another benefit of the graph-based GLogue is that the searching of an optimal plan can be reduced to a variant of shortest path problem: the optimal plan of p is a shortest "path" that has the smallest cost regarding Equation 1, from the base pattern (a single vertex) to p. An example is highlighted in Figure 5. We first assume that the queried pattern and all its sub-patterns are present in the GLogue. The process is shown in Algorithm 1.

The optimizer first builds the pattern from the `PatternDesc` that is compiled from a Gremlin query (line 2), and then organizes all *induced* subgraphs of the queried pattern by levels (line 3). Note that the use of induced subgraphs is key to ensuring worst-case optimality of the computed plan [29]. The `searchPlan` function is then launched for each pattern (line 7). We now consider processing a pattern p in the `searchPlan` function. Before searching for the plan for p, the optimal plans for all its subgraphs in the lower level must have already been computed in the `PlanMap` (line 12,16). We use the graph interface of `getEdges` in line 11 to obtain all sub-patterns in the lower level that connect to the current pattern p. Depending on whether the edge stands for a vertex expansion or binary join, the new plan will be accordingly computed in line 14 and 17. As long as

the new cost (line 18) is smaller than a previous value, the plan and its cost will be updated. The optimal plan in the GLogue can be cached to avoid re-computation. For example, in Figure 5, the cached optimal plan for computing p_9 is highlighted in red, which is the worst-case optimal plan in Figure 4(c).

Handling Pattern Miss. We discuss how to process the queried pattern p when it has not yet been recorded in the GLogue. First of all, given two *induced* subgraphs p_1, p_2 of p with $E_p = E_{p_1} \cup E_{p_2}$, by assuming the independent presence of p_1 and p_2 in the graph, we can compute the frequency of p as follows

$$\mathscr{F}(p) = \text{Avg}_{p_1,p_2} \frac{\mathscr{F}(p_1) \times \mathscr{F}(p_2)}{\mathscr{F}(p_1 \cap p_2)}, \tag{5}$$

where $p_1 \cup p_2$ denotes a pattern formed by the common parts of p_1 and p_2. Equation 5 can be recursively called in case p_1 and p_2 are not present. Then in line 11, instead of calling `getEdges` of the GLogue, we simply enumerate all p_s that can potentially expand to p, either via vertex expansion or binary join. The remaining process naturally follows Algorithm 1. In Figure 5, p_{11} is a pattern missing from the GLogue, which can either be expanded from p_{10}, or joined from p_7 and p_8. Thus, p_7, p_8 and p_{10} can all be p_s in line 11.

5.5 Dataflow Embedding

Given the optimal execution plan $\mathsf{Plan}(p)$, one last step is to embed the execution plan into a `DataflowDesc`. Following the operations of $\mathsf{Plan}(p)$, there must be some base patterns (single vertex) that are not target patterns in any operation. We encode these base patterns as `GetV` and then embed them into `Source` operators. For $\mathsf{Join}(\{p_{s_1}, p_{s_2}\} \to p_t)$, a `Join` operator is installed, which are connected by the operators that computes p_{s_1} and p_{s_2}, and the keys of the `Join` operator are set to the vertex tags of $V_{p_{s_1}} \cap V_{p_{s_2}}$.

There are two cases for hanlding a vertex expansion $\mathsf{Expand}(p_s \to p_t)$. If p_t has only one more edge than p_s, the vertex expansion will be transformed into a pair of `FlatMap(GetE)` and `Map(GetV)`, which has been discussed in Section 4.2. Otherwise, suppose p_t has $k > 1$ more edges than p_s. The execution will be decomposed into three phases, namely *attachment*, *intersection* and *unwrapping*, as shown in Figure 4(c). In the attachment phase, a `Map(GetE)` operator is installed, which tells the runtime to attach the adjacent edges of the given vertex as a *set*. The intersection phase handles $k-1$ consecutive intersection operations, while each intersection is achieved by a `Map(GetE)` that instructs computing the common edges between the existing set and the current adjacent edges. The last unwrapping phase uses a `FlatMap(GetV)` to unwrap the neighbors into discrete elements.

```
impl MapFunction for GetE { vtag,tag,label,dir } {
  fn map(&self, mut datum: GRecord) -> GRecord {
    let v = datum.get(self.vtag)?;
    // edge tag already present, do intersection
    if let Some(set) = dataum.get_mut(self.tag) {
        set.intersect(to_set(
          G.get_edges(  // G is a graph handle
            v.get_id(), self.label, self.dir))
        );
    } else { // not present, do attachment
        datum.insert(self.tag, to_set(
          G.get_edges(
            v.get_id(), self.label, self.dir));
        );
    }
    return datum;
  }
}
```

Figure 6: Code generation of `Map(GetE)`.

Example 5.2. In Figure 4(c), let's consider a mapping f that matches (v_1, v_2) before entering the process of vertex expansion. The attachment phase first maps f into $(f \mid set := \mathsf{Nbr}(f(v_1)))$ by directly attaching the adjacent edges (we reuse the notation of neighbors) as a set. In the phase of intersection, the set is updated by intersecting with the current neighbors of $f(v_2)$, as $(f \mid set := set \cap \mathsf{Nbr}(f(v_2)))$. Finally, the *set* is unwrapped into discrete vertices to match v_3.

6 System Implementation

We have implemented the frontend components including pattern parser, plan optimizer and GLogue manager in Java, to easily connect with Tinkerpop's Java runtime. The backend components are implemented in Rust to be compatible with the underlying dataflow engine, GAIA [38]. The GAIA engine runs n executors in the cluster, and each executor further forks working threads for parallel processing. The frontend and backend components of the system are bridged via the RPC services.

We implement `GRecord` to record a mapping of a pattern, which is essentially a `Map` with the key as pattern's tag (Section 4), and the value that is an `Object` to either encode a vertex, an edge, or a set of vertices/edges. Consequently, a collection of `GRecords` serve as the input and output data of all operators of the GAIA engine. Initially, the executors will load corresponding vertices as `GRecords` according to the graph partition. In the following computation, we can use the `Repartition` primitive of GAIA to reshuffle the data as needed. For example, a vertex v will be loaded by the executor numbered as "v.Id % #partitions". Moreover, in order to get adjacent edges from the matched vertices tagged as v, we can `Repartition` the `GRecords` according to the field of v.

6.1 The Dataflow Plugin

The dataflow plugin is key to making the `DataflowDesc` executable (Section 4) on the dataflow engine, which is

a native library consisted of the generated code for operators in `DataflowDesc` and a job assembler for assembling the GAIA job.

We perform code generation for all possible operators in `DataflowDesc`, and co-compile the generated code with GAIA. In Figure 6, we show the generated code of `Map(GetE)` that fulfils the phases of attachment and intersection for vertex expansion (Section 5.4). Note that in the `Map` operator for attachment in Figure 4(c), the system assigns a "_t" (as temporary) tag in the `GetE`, which will cause the adjacent edges maintaining as a set in the "_t" field of a `GRecord`. In the `Map` operator for intersection, as the "_t" field must present, it does an intersection between the existing set and the current adjacent edges.

The job assembler, after receiving the `DataflowDesc`, attempts to assemble the GAIA job. Basically, it will install the corresponding GAIA operator for each operator in the `DataflowDesc`. For example, a `Map(GetE)` will be installed as a `Map` operator with the generated code in Figure 6. The job assembler is also responsible for installing the `Repartition` primitive in case that data shuffling is needed.

7 Evaluation

7.1 Setup

Datasets. We base the evaluation on Linked Data Benchmark Council (LDBC)'s social network benchmark [30], which is the only publicly available resource that provides the scale we target in this work. As shown in Table 1, 5 datasets are generated using LDBC data generator, where G_{sf} denotes the graph generated with scale factor sf. The largest graph G_{1000} consumes around 2TB on disk and roughly 6TB aggregated memory in the cluster. The default sparsifying rate $\gamma = 100\frac{|E_{G^*}|}{|E_G|}\%$ is given for each graph. With γ, we set the memory constraint $M = \gamma|E_G|$ (Section 5.2) for graph sparsification. We by default construct the GLogue with level = 3 using the corresponding sparsified graph.

Table 1: The LDBC datasets.

Graph	$\|V\|$	$\|E\|$	Size	γ
G_1	3M	17M	1.5GB	100%
G_{30}	89M	541M	40GB	1%
G_{100}	283M	1,754M	156GB	0.1%
G_{300}	817M	5,269M	597GB	0.1%
G_{1000}	2,687M	17,789M	1,960GB	0.03%

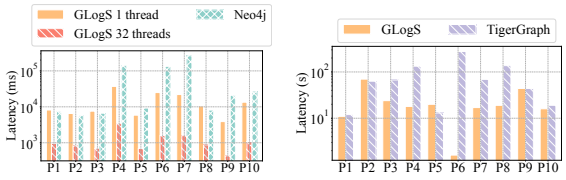
Queries. On the basis of the business intelligence (BI) workloads from LDBC benchmark, we've manually constructed 10 queries, denoted as p_1 to p_{10}, for evaluation. Details of the construction of these queries and their execution plans are in Section A.2. These queries have sufficient variance, ranging from simple triangle patterns to complex patterns like p_9 that contains 7 vertices and 9 edges. We will specify predicates corresponding to the BI workloads for p_4 to p_{10} on G_{100}, G_{300} and G_{1000}, to prevent the tests from running unnecessarily long. The LDBC benchmark driver has been modified to run each queries 5 times from a set of randomly selected parameters. Average query latency is reported.

Systems. We compare GLogS with Neo4j [33] and TigerGraph [18], two potential systems for iGPM (Section 2). We directly use the execution plans of Neo4j. By default, GLogS runs the optimal execution plan of a query p, which are derived from the GLogue (on each sparsified graph) specifically constructed from p. As TigerGraph does not have an optimizer, on the one hand, we used the queries generated by its graph studio [47], on the other hand, we manually wrote the queries according to the optimal plans of GLogS. Note that we did not include the results of our base system, GAIA [38], as it could not terminate in reasonable time in most of our large-scale tests. On the one hand, GAIA has optimized graph workloads by utilizing a breadth-first/depth-first hybrid scheduling and memory-bounded execution model, enabling it to handle considerable amounts of data without overflowing the memory. In fact, our GLogS has benefited from GAIA in handling large-scale data (Section 7.3). On the other hand, GAIA lacks proficiency in executing GPM queries efficiently, primarily due to the fact that GAIA must comply with Gremlin's imperative traversal which conforms to a sub-optimal EdgeJoin execution plan [29] that sequentially joins edges.

We use the default system configurations of Neo4j and TigerGraph. For GLogS, we have measured the differences in the operations of vertex expansion and binary join, as well as the communication and computation cost, which allows us to set $\alpha_j = 60$ and $\alpha_{ve} = 1$ in Equations 2 and 3, respectively. We deploy a cluster for the evaluation that contains one frontend server and up to 16 backend servers. Each server configures two 24-core Intel(R) Xeon(R) Platinum 8163 CPUs at 2.50GHz and a 512GB RAM. The servers are connected to an EDR 25 Gbps InfiniBand network, which can scale deterministically and achieve full bisection bandwidth. If not mentioned, we will use 32 threads on each server for GLogS (some threads are reserved for communication and system calls) as suggested by GAIA authors [38]. TigerGraph will use all threads as recommended.

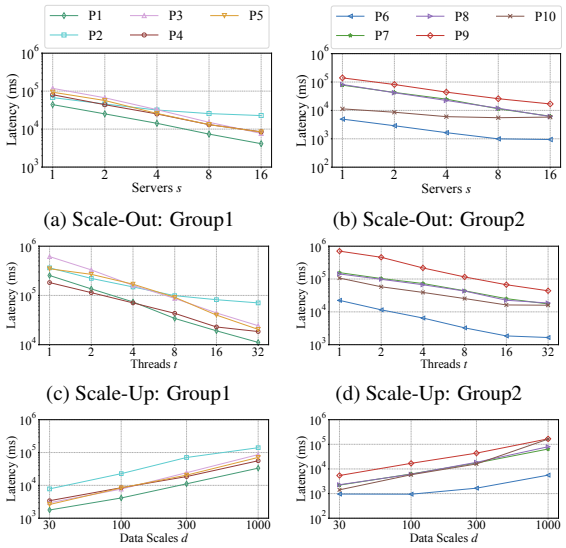
7.2 Compare with Alternative Systems

We first compare GLogS with Neo4j on a single machine, using the smallest graph G_1 to allow Neo4j processing all queries in a reasonable time. The query latencies of both systems are shown in Figure 7a. GLogS with a single thread still performs better than Neo4j for most queries, with 4.4× speedup. After using 32 threads, GLogS out-



(a) GLogS v.s. Neo4j on G_1 (b) GLogS v.s. TigerGraph on G_{300}

Figure 7: Compare with alternative systems.



(a) Scale-Out: Group1 (b) Scale-Out: Group2

(c) Scale-Up: Group1 (d) Scale-Up: Group2

(e) Data-Scale: Group1 (f) Data-Scale: Group2

Figure 8: Scalability Experiments

performs Neo4j by an average of 51× for all queries.

The tests against TigerGraph are performed on G_{300} using 16 machines. While benefiting from the optimal execution plans of GLogS, TigerGraph still performs 59% slower on average. Note that the results obtained from TigerGraph's graph studio are not reported as they exceed the time limit (one hour) in all tests. Additionally, TigerGraph requires the installation of queries via native-code compilation before they can be run, with installation times ranging from 1 to 3 minutes. In contrast, GLogS does not have this overhead because of the design of dataflow plugin (see Section 6.1). The time required to compile and optimize all queries in GLogS is less than 1 millisecond, which is insignificant compared to the query execution time. This demonstrates GLogS's advantage in usability for iGPM.

7.3 Scalability

It's important to test the scalability on GPM workload given its nature of irregularity [14, 29, 54]. The results are in Figure 8, where the queries are split into two groups based on their latency for clear illustration.

Scale-Out. We vary the server number as 1, 2, 4, 8, 16 and run all queries on G_{100}. Note that G_{100} is the largest graph that can reside in the main memory of a single server. The results are reported in Figure 8a and Figure 8b. Most queries scale well, with up to 15× (average 6×) performance gain from one machine to 16.

Scale-Up We use 16 servers and vary the number of working threads on each server from 1 to 32. The results on G_{300} are shown in Figure 8c and Figure 8d. We see an improvement in runtime of up to 23× (average of 10×) when increasing the number of threads from 1 to 32. A common trend in both scale-out and scale-up tests is that some queries, such as p_2 and p_{10}, scale less significantly when using more working threads. This phenomenon is not unique to our system [14, 38] and is mostly due to the sensitivity of GPM workloads to data skew [14]. It's a future work to further address this issue.

Data-Scale Using 16 servers, we run all queries on the graphs of G_{30}, G_{100}, G_{300} and G_{1000}. The results are reported in Figure 8e and Figure 8f. As the graphs become larger, most queries demonstrate an almost linear trend in performance degradation, except for p_6 and p_{10}. As for p_6, its execution time only tripled from G_{30} to G_{1000}, because it is a short-running query that visits a small part of the graph. However, for p_{10}, its performance degrades by 100× from G_{30} to G_{1000}. This is likely because the execution plan for p_{10} is the only plan that involves a join operator, which maintains a hashmap for the "build" component of the join. When processing a large volume of data, a hashmap lookup can become slower because many entries may have been mapped to the same bucket. Despite this, the plan with the join operator still performs much better than the one without it.

In summary, GLogS exhibits excellent scalability in the test, which we attribute to both the well-designed optimizer and GAIA's graph-specific optimizations.

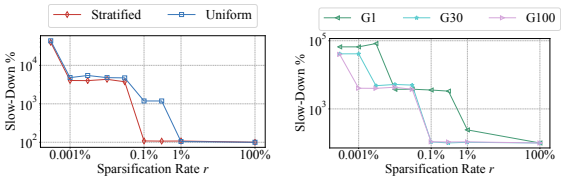
7.4 Plan Optimization

In this experiment, we will study the impact of high-order statistics and graph sparsification on plan optimization for all queries. While running a query, we obtain the time t using the optimal execution plan, and the time t' using the computed execution plan in a certain context. We report the slow-down rate as $100\frac{t'}{t}\%$. For convenience, we run all tests in a single server.

Table 2: The effectiveness of high-order statistics.

	level=2	level=3	level=4
Slow-down (%)	966	245	243
Generation Time(s)	6	55	1664
Memory Usage(GB)	2	3	105
# Patterns	34	248	4164

High-Order v.s. Low-Order. To study the effectiveness of high-order statistics, we construct the GLogue of level 2, 3 and 4 for G_1, and try to evaluate the average slow-down rate of all queries while using the execution plans



(a) Uniform v.s. Stratified (b) Vary graphs.

Figure 9: Impact of sparsification on plan optimization.

computed from these GLogue. Here, we use the unsparsified G_1 to rule out potential impact from sparsification. The results are shown in Table 2. When increasing the level of GLogue from 2 (low-order statistics only) to 3, the performance of queries improves by 4× on average. The shows that high-order statistics can contribute to deriving better execution plans for GPM queries. Moreover, the performance of queries remains almost unchanged when we increase GLogue's level from 3 to 4, but the time and memory consumption for constructing the 4-level GLogue increase significantly. Given that the GLogue will be further updated from frequently queried patterns, we suggest that the initial construction of 3-level GLogue is sufficient.

Uniform v.s. Stratified. This test verifies the advantage of the proposed stratified sparsification over the uniform alternative. As shown in Figure 9a, we apply both methods on G_{100}, and report the average slow-down rate for all queries using the execution plans computed from the sparsified graphs of various sparsifying rate ranging from 0.001% to 100%. Stratified sparsification performs much better than the uniform alternative, as it has resulted in a better execution plan at a lower sparsifying rate and, at the same sparsifying rate, it has achieved a lower slow-down. With statified sparsification, the graph can be sparsified 10× more edges, on which the optimizer can still derive the optimal execution plans.

Vary Graphs. To verify whether we can use larger sparsifying rate on larger graph, we sparsify three graphs G_1, G_{30} and G_{100} (G_{300} and G_{1000} are too large to process in a single server) using different rates, and report the average slow-down of all queries from the resulting plans in Figure 9b. Clearly, larger graphs can be sparsified at a lower rate, while still rendering good execution plans. For example, the performance of queries on G_{100} and G_{30} only notably downgrades when $\gamma < 0.1\%$. In comparison, the downgrading point of G_1 are $\gamma < 1\%$. Furthermore, while sparsified to 0.001%, the resulting plans from G_{100} slow down by roughly 50×, but those from G_{30} and G_1 downgrade by over 500×.

8 Related Work

GPM Algorithms. Ullmann proposed the first backtracking algorithm [50] for GPM, based on which many optimizations have been proposed, such as tree indexing [41], symmetry breaking [25] and compression [13]. As it's hard to parallelize the backtracking algorithm, join-based algorithms, such as binary-join algorithms [28, 29, 43], have been developed in the distributed context. Aware that binary-join algorithms cannot guarantee worst-case optimality, [4] implemented the worst-case-optimal join algorithm [35] for solving GPM. A hybrid mechanism [1, 32, 54] has been further explored to combine the advantage of binary join and worst-case optimal join. The above algorithms all rely on low-order statistics to devise execution plans. In order to improve cost estimation, [32] proposed to leverage the high-order statistics of the graph to compute execution plans for GPM. These algorithmic approaches are lacking essential system components needed to solve iGPM.

Query Languages and Graph Databases. GPM lies at the core of the query languages of Gremlin [39], Cypher [22], G-Core [5], PGQL [51] and GSQL [18]. These languages have been widely adopted in graph databases and systems. Tinkerpop [7] uses the Gremlin language to express graph traversal and pattern matching. Neo4j [33] is one of the most popular graph databases that uses Cypher as the query language. Gremlin-enabled JanusGraph [27], Orient DB [36] and Neptune [9] store graph data in distribution, but they adopt a sequential computing engine and can still suffer from scalability issue [38]. Targeting large scale, GAIA [38] has been developed to compile Gremlin traversal queries into a distributed dataflow program. However, the imperative Gremlin traversal cannot guarantee worst-case optimality, and it requires users to manually tune the execution. TigerGraph [18] is a distributed graph database, that uses the GSQL query language. However, the lack of an automatic optimizer greatly limits its usability for iGPM.

9 Conclusion

We've presented the GLogS system in this paper to solve the iGPM, meeting the requirements of performance, usability and scalability. GLogS allows users to interactively submit declarative GPM queries. With the worst-case optimality and high-order statistics, we've implemented an optimizer in GLogS that can automatically derive optimal execution plans for arbitrary GPM queries. Furthermore, on top of an existing distributed dataflow engine, GLogS is capable of being deployed in a large cluster to handle large-scale real-life graphs.

Acknowledgments

We sincerely thank our shepherd Călin Iorgulescu and the anonymous reviewers for their insightful comments. This work was supported by Alibaba Innovative Research (AIR) Program.

References

[1] ABERGER, C. R., LAMB, A., TU, S., NÖTZLI, A., OLUKOTUN, K., AND RÉ, C. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS) 42*, 4 (2017), 1–44.

[2] AGENSGRAPH. `https://bitnine.net/`. [Online; accessed 20-October-2022].

[3] ALON, N., DAO, P., HAJIRASOULIHA, I., HORMOZDIARI, F., AND SAHINALP, S. C. Biomolecular network motif counting and discovery by color coding. *Bioinformatics 24*, 13 (2008), i241–i249.

[4] AMMAR, K., MCSHERRY, F., SALIHOGLU, S., AND JOGLEKAR, M. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow. 11*, 6 (oct 2018), 691–704.

[5] ANGLES, R., ARENAS, M., BARCELO, P., BONCZ, P., FLETCHER, G., GUTIERREZ, C., LINDAAKER, T., PARADIES, M., PLANTIKOW, S., SEQUEDA, J., VAN REST, O., AND VOIGT, H. G-core: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, Association for Computing Machinery, p. 1421–1432.

[6] ANGLES, R., ARENAS, M., BARCELÓ, P., HOGAN, A., REUTTER, J., AND VRGOČ, D. Foundations of modern query languages for graph databases. *ACM Comput. Surv. 50*, 5 (sep 2017).

[7] APACHE TINKERPOP. `http://tinkerpop.apache.org/`. [Online; accessed 20-October-2022].

[8] ARANGODB. `https://www.arangodb.com/`. [Online; accessed 20-October-2022].

[9] AWS NEPTUNE. `https://aws.amazon.com/neptune/`. [Online; accessed 20-October-2022].

[10] BABAI, L., AND LUKS, E. M. Canonical labeling of graphs. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1983), STOC '83, Association for Computing Machinery, p. 171–183.

[11] BATSON, J., SPIELMAN, D. A., SRIVASTAVA, N., AND TENG, S.-H. Spectral sparsification of graphs: theory and algorithms. *Communications of the ACM 56*, 8 (2013), 87–94.

[12] BENSON, A. R., GLEICH, D. F., AND LESKOVEC, J. Higher-order organization of complex networks. *Science 353* (2016), 163–166.

[13] BI, F., CHANG, L., LIN, X., QIN, L., AND ZHANG, W. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1199–1214.

[14] CHEN, X., ET AL. Efficient and scalable graph pattern mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (2022), pp. 857–877.

[15] CHEN, Z., CHEN, L., VILLAR, S., AND BRUNA, J. Can graph neural networks count substructures? *Advances in neural information processing systems 33* (2020), 10383–10395.

[16] CYPHER QUERY LANGUAGE. `https://neo4j.com/developer/cypher/`. [Online; accessed 20-October-2022].

[17] DESHPANDE, M., KURAMOCHI, M., WALE, N., AND KARYPIS, G. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering 33*, 8 (2005), 1036–1050.

[18] DEUTSCH, A., XU, Y., WU, M., AND LEE, V. Tigergraph: A native mpp graph database. *arXiv preprint arXiv:1901.08248* (2019).

[19] ESFAHANI, M. S., AND DOUGHERTY, E. R. Effect of separate sampling on classification accuracy. *Bioinformatics 30*, 2 (2014), 242–250.

[20] FLAKE, G. W., LAWRENCE, S., GILES, C. L., AND COETZEE, F. M. Self-organization and identification of web communities. *Computer 35*, 3 (2002), 66–70.

[21] FOR LANGUAGE RECOGNITION, A. T. `https://www.antlr.org/`. [Online; accessed 20-October-2022].

[22] FRANCIS, N., GREEN, A., GUAGLIARDO, P., LIBKIN, L., LINDAAKER, T., MARSAULT, V., PLANTIKOW, S., RYDBERG, M., SELMER, P., AND TAYLOR, A. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 1433–1445.

[23] GAÜZÈRE, B., BRUN, L., AND VILLEMIN, D. Graph kernels in chemoinformatics. In *Quantitative Graph TheoryMathematical Foundations and Applications*, M. Dehmer and F. Emmert-Streib, Eds. CRC Press, 2015, pp. 425–470.

[24] GROCHOW, J. A., AND KELLIS, M. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Research in Computational Molecular Biology* (Berlin, Heidelberg, 2007), T. Speed and H. Huang, Eds., Springer Berlin Heidelberg, pp. 92–106.

[25] HAN, W.-S., LEE, J., AND LEE, J.-H. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, Association for Computing Machinery, p. 337–348.

[26] HU, Y., JI, S., JIN, Y., FENG, L., STANLEY, H. E., AND HAVLIN, S. Local structure can identify and quantify influential global spreaders in large scale social networks. *Proceedings of the National Academy of Sciences 115*, 29 (2018), 7468–7472.

[27] JANUSGRAP. `https://janusgraph.org/`. [Online; accessed 20-October-2022].

[28] LAI, L., QIN, L., LIN, X., AND CHANG, L. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment 8*, 10 (2015), 974–985.

[29] LAI, L., QING, Z., YANG, Z., JIN, X., LAI, Z., WANG, R., HAO, K., LIN, X., QIN, L., ZHANG, W., ET AL. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment 12*, 10 (2019), 1099–1112.

[30] LDBC SOCIAL NETWORK BENCHMARK. `https://ldbcouncil.org/benchmarks/snb/`. [Online; accessed 20-October-2022].

[31] LOUKAS, A., AND VANDERGHEYNST, P. Spectrally approximating large graphs with smaller graphs. In *International Conference on Machine Learning* (2018), PMLR, pp. 3237–3246.

[32] MHEDHBI, A., AND SALIHOGLU, S. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076* (2019).

[33] MILLER, J. J. Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA* (2013), vol. 2324.

[34] NEO4J EXECUTION PLAN. `https://neo4j.com/docs/cypher-manual/current/execution-plans/`. [Online; accessed 20-October-2022].

[35] NGO, H. Q., PORAT, E., RÉ, C., AND RUDRA, A. Worst-case optimal join algorithms. *Journal of the ACM (JACM) 65*, 3 (2018), 1–40.

[36] ORIENT DB. `hhttps://orientdb.org/`. [Online; accessed 20-October-2022].

[37] PRŽULJ, N., CORNEIL, D. G., AND JURISICA, I. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics 22*, 8 (2006), 974–980.

[38] QIAN, Z., MIN, C., LAI, L., FANG, Y., LI, G., YAO, Y., LYU, B., ZHOU, X., CHEN, Z., AND ZHOU, J. GAIA: A system for interactive analysis on distributed graphs using a High-Level language. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 321–335.

[39] RODRIGUEZ, M. A. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (2015), pp. 1–10.

[40] SANEI-MEHRI, S.-V., SARIYUCE, A. E., AND TIRTHAPURA, S. Butterfly counting in bipartite networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2018), pp. 2150–2159.

[41] SHANG, H., ZHANG, Y., LIN, X., AND YU, J. X. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow. 1*, 1 (aug 2008), 364–375.

[42] SPIELMAN, D. A., AND TENG, S.-H. Spectral sparsification of graphs. *SIAM Journal on Computing 40*, 4 (2011), 981–1025.

[43] STEINBRUNN, M., MOERKOTTE, G., AND KEMPER, A. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal 6*, 3 (1997), 191–208.

[44] SUN, S., SUN, X., CHE, Y., LUO, Q., AND HE, B. Rapidmatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment 14*, 2 (2020), 176–188.

[45] TEIXEIRA, C. H., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 425–440.

[46] THE OPEN SOURCE PAGE OF GLOGS. `https://github.com/MeloYang05/GLogS-Artifact`. [Online; accessed 7-June-2023].

[47] TIGERGRAPH GRAPHSTUDIO. `https://docs.tigergraph.com/gui/current/graphstudio/build-graph-patterns/visual-query-builder-overview`. [Online; accessed 20-October-2022].

[48] TRUTH BEHIND NEO4J'S "TRILLION" RELATIONSHIP GRAPH. `https://www.tigergraph.co.jp/blog/truth-behind-neo4js-trillion-relationship-graph/`. [Online; accessed 20-October-2022].

[49] TSOURAKAKIS, C. E., DRINEAS, P., MICHELAKIS, E., KOUTIS, I., AND FALOUTSOS, C. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining 1*, 2 (2011), 75–81.

[50] ULLMANN, J. R. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM) 23*, 1 (1976), 31–42.

[51] VAN REST, O., HONG, S., KIM, J., MENG, X., AND CHAFI, H. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2016), GRADES '16, Association for Computing Machinery.

[52] VERMA, S., LESLIE, L. M., SHIN, Y., AND GUPTA, I. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow. 10*, 5 (jan 2017), 493–504.

[53] XU, K., HU, W., LESKOVEC, J., AND JEGELKA, S. How powerful are graph neural networks? In *International Conference on Learning Representations* (2019).

[54] YANG, Z., LAI, L., LIN, X., HAO, K., AND ZHANG, W. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 2049–2062.

[55] YOU, J., GOMES-SELMAN, J. M., YING, R., AND LESKOVEC, J. Identity-aware graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2021), vol. 35, pp. 10737–10745.

A Appendix

A.1 Graph Sparsification

In the paper, we adopt the stratified sparsification [19] that treats each type of edges as an independent stratum, and assign each stratum an individual sparsification ratio. For clarity, we use v (or e) and u (or ε) to denote a vertex (or edge) in the pattern and graph, respectively. Moreover, given an edge e (or ε), we denote its label as $L(e)$. Let $\{1,2,\ldots,l\} \subset \mathbb{N}^+$ be the domain of edge labels without loss of generality, and $\{s_1,s_2,\ldots,s_l\}$ be the frequencies of the edges with the given label in the graph G. Then we define the sparsification ratios as $\Omega = \{\rho_1,\rho_2,\ldots,\rho_l\}$, where ρ_i denotes the ratio for the stratum of edges with label i.

For two random variables X, X', we denote $\mathbb{E}[X], \mathrm{Var}[X]$ as the expected value and variance of X, and $\mathrm{Cov}[X,X']$ the covariance of X and X'. Given that we only eliminate edges in the sparsification process, we assume that the vertex set retains after sparsification, namely $V_G = V_{G^*}$.

A.1.1 The Proof of Lemma 6.1

Proof. Let $E_G(u_1,u_2)=1$ indicate the existence of edge (u_1,u_2) in the graph G and 0 otherwise. We formulate that

$$\mathscr{F}_{G^*}(p)=\underbrace{\sum_{(u_1,u_2,\ldots,u_{|V_p|})\in V_{G^*}^{|V_p|}}}_{\text{For each possible subgraph,}}\underbrace{\prod_{\substack{e\in E_p,\\ e=(v_j,v_k)}}E_{G^*}(u_j,u_k)}_{\text{verify the existence}}, \tag{6}$$

and obtain

$$\begin{aligned}&\mathbb{E}[\mathscr{F}_{G^*}(p)]\\ =&\sum_{(u_1,u_2,\ldots,u_{|V_p|})\in V_{G^*}^{|V_p|}}\prod_{\substack{e\in E_p,\\ e=(v_j,v_k)}}\rho_{L(e)}\times E_G(u_j,u_k)\\ =&\prod_{e\in E_p}\rho_{L(e)}\sum_{(u_1,u_2,\ldots,u_{|V_p|})\in V_{G^*}^{|V_p|}}\prod_{\substack{e\in E_p,\\ e=(v_j,v_k)}}E_G(u_j,u_k),\end{aligned}$$

Since $V_G=V_{G^*}$, we have

$$\mathbb{E}[\mathscr{F}_{G^*}(p)]=\prod_{e\in E_p}\rho_{L(e)}\mathscr{F}_G(p).$$

Thus, the lemma holds.

A.1.2 The Optimization Problem for the Stratified Sparsification

Let M denote a memory constraint to ensure that the sparsified graph can reside in the frontend machine. We first consider a specific pattern p, and later generalize to an arbitrary pattern. We formulate the stratified sparsification as an optimization problem as:

$$\begin{aligned}&\arg\min_{\Omega}\mathrm{Var}[\widetilde{\mathscr{F}(p)}]\\ &\text{s.t.}\sum_{i=1}^{l}s_i\rho_i\le M.\end{aligned} \tag{7}$$

Given an ordered set of vertices $S=(u_1,u_2,\ldots,u_{|V_p|})\in V_G^{|V_p|}$ from the graph G, we denote $\mathbb{1}(f_G(p)|S)$ to indicate whether there is a subgraph with S in G that can match the pattern p. If $\mathbb{1}(f_G(p)|S)=1$, in other words, S must be mapping of p in G, we directly use $f_G(p)|S$ to represent the matched subgraph regarding S. Note that the notations will be applied to both the eMap and G^* in the following. For example, given a vertex set S (same vertex set in both eMap and G^*), if $\mathbb{1}(f_G(p)|S)=1$ but $\mathbb{1}(f_{G^*}(p)|S)=0$, we know that some edge in the matched subgraph has been eliminated during sparsification. With the indicator, we have

$$\mathscr{F}_{G^*}(p)=\sum_{S\in V_{\texttt{eMap}}^{|V_p|}}\mathbb{1}(f_{G^*}(p)|S).$$

Therefore,

$$\begin{aligned}&\mathrm{Var}[\mathscr{F}_{G^*}(p)]\\ =&\sum_{S_1,S_2\in V_G^{|V_p|}}\mathrm{Cov}\left[\mathbb{1}(f_{G^*}(p)|S_1),\mathbb{1}(f_{G^*}(p)|S_2)\right]\\ =&\sum_{S_1,S_2\in V_G^{|V_p|}}(\mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_1)\mathbb{1}(f_{G^*}(p)|S_2]\\ &-\mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_1)]\times\mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_2)].\end{aligned} \tag{8}$$

We observe that the covariance in Equation 8 must be zero in either of the following case.

- S_1 or S_2 cannot form a mapping of p in G, i.e., $\mathbb{1}(f_{\texttt{eMap}}(p)|S_1)=0$ or $\mathbb{1}(f_{\texttt{eMap}}(p)|S_2)=0$. In this case, the matched subgraph must not exist in G^*, leading to $\mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_1)]=0$ or $\mathbb{E}[\mathbb{1}(f_{G^*}(p)|S_2)]=0$.
- $S_1\cap S_2=\emptyset$, namely the two sets are disjoint.

Therefore, we only need to study the two sets S_1 and S_2, such that $\mathbb{1}(f_{\texttt{eMap}}(p)|S_1)=1$ and $\mathbb{1}(f_{\texttt{eMap}}(p)|S_2)=1$, and $S_1\cap S_2\neq\emptyset$. In this case, $f_{\texttt{eMap}}(p)|S_1$ and $f_{\texttt{eMap}}(p)|S_2$ may share either none, one, or more than one common edges. The trivial case of sharing no edge results in zero variance. For other cases, we empirically studied their occurrences while matching all benchmark queries on G_1. We found that the case of sharing one single edge occurs much more frequently than that of sharing multiple edges. Let the common edge be ε, which must be matched by a pattern edge $e'\in E_p$. According to Equation 8, the covariance becomes

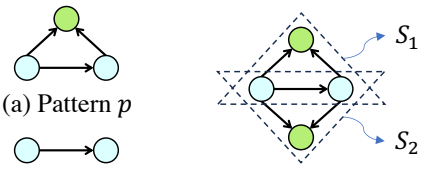
$$\begin{aligned}&\mathrm{Cov}\left[\mathbb{1}(f_{\texttt{eMap}}(p)|S_1),\mathbb{1}(f_{\texttt{eMap}}(p)|S_2)\right]\\ =&\left(\prod_{e\in E_p}\rho_{L(e)}\right)^2*\left(\rho_{L(e')}^{-1}-1\right).\end{aligned} \tag{9}$$

We then group these pairs by the labels of e', which eliminates e' in Equation 9 and transforms Equation 8 into

$$\begin{aligned}&\mathrm{Var}[\mathscr{F}_{G^*}(p)]\\ \approx&\left(\prod_{e\in E_p}\rho_{L(e)}\right)^2*\sum_{e\in E_p}\left(\rho_{L(e)}^{-1}-1\right)\lambda_{\texttt{eMap}}(p|e),\end{aligned} \tag{10}$$

where $\lambda_{\texttt{eMap}}(p|e)$ denotes the number of pairs of S_1 and S_2 in eMap whose common edge is matched by e in pattern p.

Figure 10 demonstrates a pair of S_1 and S_2 that matches a triangle pattern, as an example. In addition, matched subgraphs $f_{\texttt{eMap}}(p|S_1)$ and $f_{\texttt{eMap}}(p|S_2)$ share a common edge in the example. We observe that $f_{\texttt{eMap}}(p|S_1)$ and $f_{\texttt{eMap}}(p|S_2)$ together form a new pattern,



(a) Pattern p

(b) Common edge e (c) A pair of S_1 and S_2

Figure 10: Example of a pair of S_1 and S_2 with shared edge e.

which is a mirror symmetric structure with respect to the common edge. In terms of an edge $e \in E_p$, we denote such a mirror pattern as p_e^+. It's clear that $\lambda_{\texttt{eMap}}(p|e)$ is equal to the frequency of pattern p_e^+ in eMap, namely $\lambda_{\texttt{eMap}}(p|e) = \mathscr{F}_G(p_e^+)$. Combining with the Equation 4 in the paper, we have

$$\lambda_{\texttt{eMap}}(p|e) = \mathscr{F}_G(p_e^+) \approx \frac{\mathscr{F}_G(p) \times \mathscr{F}_G(p)}{\mathscr{F}_G(e)} = \frac{\mathscr{F}_G(p)^2}{s_{L(e)}}, \tag{11}$$

Combining the definition of $\widetilde{\mathscr{F}(p)}$, Equation 10, and Equation 11, we obtain

$$\text{Var}[\widetilde{\mathscr{F}(p)}] \approx \sum_{e \in E_p} \left(\rho_{L(e)}^{-1} - 1\right) \frac{\mathscr{F}_G(p)^2}{s_{L(e)}},$$

Note that $\mathscr{F}_G(p)$ can be treated as a constant value in the optimization problem. Consequently, we can rephrase the optimization problem in Equation 7 as

$$\begin{aligned} &\arg\min_{\Omega} \sum_{e \in E_p} \left(\rho_{L(e)} s_{L(e)}\right)^{-1}, \\ &\text{s.t.} \sum_{i=1}^{l} s_i \rho_i \leq M. \end{aligned} \tag{12}$$

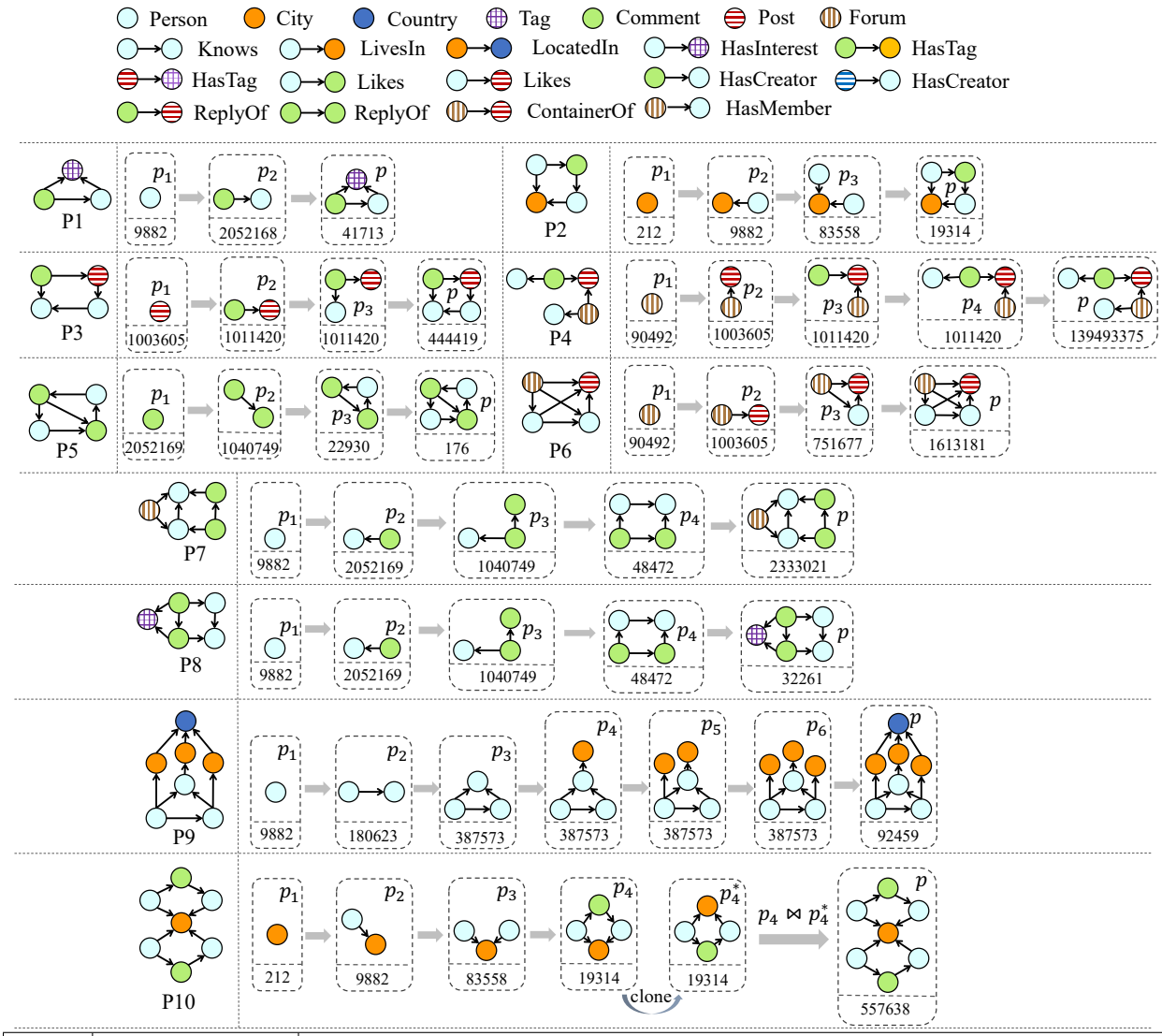
Till now, the optimization problem only considers a specific pattern p. For generalizing to an arbitrary pattern, we construct a pattern p that is formed by all types of edges in the graph, and in Equation 12, we enumerate all labels rather than those in the pattern p, which becomes

$$\begin{aligned} &\arg\min_{\Omega} \sum_{i=1}^{l} \left(s_i \rho_i\right)^{-1}, \\ &\text{s.t.} \sum_{i=1}^{l} s_i \rho_i \leq M. \end{aligned}$$

The minimal variance is achieved when $\rho_i = \frac{M}{l} \times \frac{1}{s_i}$. Since the sparsification ratio ρ_i has a upper bound 1, i.e., all edges with label i are preserved, we have $\rho_i = \min(1, \frac{M}{l} \times \frac{1}{s_i})$.

A.2 Queries and Execution Plans

We reported all queries used in the experiments, along with their execution plans generated by the optimizer of GLogS, in Figure 11. In the execution plans, we also marked the corresponding intermediate pattern frequencies in the benchmark graph G_1 for the evaluation of their performance. The table in Figure 11 explains how the queries are constructed. Specifically, their main structures of all queries are extracted from LDBC [30] Business Intelligence (BI) workloads, and then modified to cover more test scenarios. Overall, the queries contains Long-Chain, Triangle, Square, 4-Clique, House that are commonly used for benchmarking GPM queries [29, 32]. Their execution plans cover both Expand and Join operators.



Query	Source	Explanation
p_1	BI-8	p_1 is extracted from BI-8
p_2	BI-5	Wedge Person → Comment → Person is extracted from BI-5. Additionally, a City vertex and two LivesIn edges are added to form a Square
p_3	BI-15	p_3 is extracted from BI-15
p_4	BI-17	p_4 is extracted from BI-17
p_5	BI-17	p_5 is extracted from BI-17
p_6	BI-4, BI-15	Some subgraphs are extracted from BI-4 and BI-15 to form a 4-Clique
p_7	BI-19, BI-4	Its right square is extracted from BI-19. In addition, a Forum hat is extracted from BI-4 and added to the square to form a House
p_8	BI-19, BI-17	Its right square is extracted from BI-19. In addition, a Tag hat is extracted from BI-17 and added to the square to form a House
p_9	BI-11	p_9 is the main structure of BI-11
p_{10}	BI-5	p_{10} consists of two p_2 by joining on the City vertex. This is designed to verify whether GLogue can generate a plan with join

Figure 11: Queries and their executions plans generated by GLogS's Optimizer

Cyclosa: Redundancy-Free Graph Pattern Mining via Set Dataflow

Chuangyi Gui[†‡], Xiaofei Liao[†], Long Zheng[†‡], Hai Jin[†]
[†]*National Engineering Research Center for Big Data Technology and System/ Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology, China*
[‡]*Zhejiang Lab, China*

Abstract

Graph pattern mining is an essential task in many fields, which explores all the instances of user-interested patterns in a data graph. Pattern-centric mining systems transform the patterns into a series of set operations to guide the exploration and substantially outperform the embedding-centric counterparts that exhaustively enumerate all subgraphs. These systems provide novel specializations to achieve optimum search space, but the inherent redundancies caused by recurrent set intersections on the same or different subgraph instances remain and are difficult to trace, significantly degrading the performance.

In this paper, we propose a dataflow-based graph pattern mining framework named Cyclosa to eliminate the above redundancies by utilizing the concept of computation similarity. Cyclosa is characterized by three features. First, it reorganizes the set operations for a pattern into a set dataflow representation which can elegantly indicate the possibility of redundancies while sustaining the optimal scheduling for high performance. Second, the dataflow-guided parallel execution engine decouples data access and computations to enable efficient results sharing. Third, the memory-friendly data management substrate can automatically manage the computation results with high reuse possibility. Evaluation of different patterns demonstrates that Cyclosa outperforms state-of-the-art pattern-centric systems GraphPi and SumPA by up to 16.28× and 5.52×, respectively.

1 Introduction

Graphs are the *de facto* paths to explore useful information in various fields, including social media analysis [1, 2], financial networks [3, 4], and bioinformatics [5]. Graph pattern mining aims to explore interesting subgraph structures according to the user-given constraints. Typical graph pattern mining applications include subgraph matching [6, 7], clique finding [8, 9], and motifs counting [10–12]. Despite the prevalence of graph pattern mining applications, they have high computational complexity and usually need hours or even days to complete [13–15].

Graph pattern mining systems have emerged in recent years to provide high performance and programmability [16–18]. A common approach is to enumerate all the subgraphs, usually under a certain depth, to check whether the subgraphs satisfy the pattern constraints, which is called the embedding-centric paradigm [16, 19]. This approach is easy to develop and parallelize. However, it results in high memory consumption and wasted computing resources due to a large number of intermediate partial instances [17, 20]. Recently, advanced graph pattern mining systems have adopted a pattern-centric paradigm to overcome inefficiencies [17]. The main idea is to use the structure information of graph patterns to filter intermediates that will not lead to a correct final match. This is achieved by transforming the graph patterns into a series of set operations and executing them in a nested loop following a matching order of pattern vertices. Each loop computes the candidates of corresponding pattern vertex, where the computation is represented as a formula of set intersections on the neighboring lists of previously matched pattern vertices based on the structural connectivity, e.g., $Cand(v_2) = N(v_0) \cap N(v_1)$ for a triangle pattern after matching an edge (v_0, v_1). In this way, only valid partial instances are produced in each loop.

Prior works propose many novel techniques to reduce the search space also the number of partial instances to be explored. AutoMine [17] provides a convenient compiler to generate optimized matching orders automatically (also the order of computations), which will significantly influence the search space. Peregrine [20] and GraphZero [21] introduce a symmetry-breaking method that filters the partial instances leading to the same final mapping by comparing vertex IDs of symmetric vertices. GraphPi [22] further explores an optimal combination of different symmetry-breaking rules and matching orders to minimize the search space. The above systems mainly optimize the mining of a single pattern. SumPA [23] further proposed an abstraction approach to reduce workloads for mining multiple patterns simultaneously.

However, a large number of intrinsic redundant computations remain in the execution of set operations even in an optimized search space, the same set intersection, e.g.,

$N(v_0) \cap N(v_1)$, repeats throughout the processing. Specifically, the redundancies can be classified into two categories, the *explicit redundancy* and *implicit redundancy*, based on whether they rely on the same subgraph instance. In explicit redundancies, one set intersection can be repeatedly used for computing different pattern vertices connected to the same subgraph instances. In implicit redundancies, the same intersection appears in computing on different subgraph instances. The redundant computations usually cost more than 80% of the runtime and severely degrade the performance. Furthermore, these explicit and implicit redundancies spread over the runtime when mining a single pattern or multiple patterns, making it difficult to trace and reuse. Existing systems follow the principle of structural equality and rewrite the exact same set formulas of different vertices into a single one to reduce part of the explicit redundancies [21–23]. However, there are still more than 60% of the redundancies remaining unsolved.

We observe two kinds of computation similarity in the set operations providing the opportunity to help identify and reuse both explicit and implicit redundancies. The first one is the static similarity which reveals the structural similarity among the operands of the set operations in a pattern, that one input operand can be reused in two operators, and the output results of these operators may have latent redundancies. The static similarity can be analyzed before execution. We propose to decouple the operands and operators of all set operations and organize them into a directed flow oriented by the connections among the inputs and outputs, which is called a *set dataflow*, to efficiently exploit the static similarity. Each node in the set dataflow is a set operand or a set operator. The explicit redundancies can be removed by keeping only unique operands in the dataflow, while the implicit redundancies between different operators can be indicated by the overlapped input source nodes of the dataflow.

The second one is the dynamic similarity which reveals the similarity among the inputs of all occurred set intersections during runtime, which can only be analyzed after execution. Specifically, we observe that a small number of high-degree vertices participate in most of the computations, which means redundant computations are concentrated in these vertices. This allows us to cache and reuse implicit redundant results by tracing these high-degree vertices. However, maintaining correctness and efficiency in the execution of the dataflow is challenging. How to manage the intermediate computation results in limited memory space is also a main difficulty.

In this paper, we present Cyclosa, a novel dataflow-based graph pattern mining system to eliminate both explicit and implicit redundancies in the pattern-centric paradigm. Specifically, Cyclosa is characterized by the following key features. First, we propose a novel set dataflow representation and an efficient constructing approach to generate the set dataflow for arbitrary patterns. It maintains optimized cost through a lightweight cost estimation model and introduces the redundancy probability to guide the appropriate reusing of results. Second, we develop a dataflow-based execution model to expose the possibility of capturing and reusing redundancies during runtime. Through the dataflow execution, the data accesses and computations are decoupled, providing the ability to maximize data reuse in parallel. Third, we design a memory-friendly data management substrate to automatically store the computation results with a high possibility of redundancy. It implements smart cache strategies according to the reuse probability of results evaluated, thus achieving a controlled memory consumption. Furthermore, the substrate can efficiently cooperate with the dataflow execution engine to provide the results requested by repeated computations. We implement a prototype of Cyclosa and achieve significant performance improvement over state-of-the-art pattern-centric graph pattern mining systems.

The key contributions of this paper are as follows:

- It introduces a set dataflow representation to explore the fine-grained computation similarity in set operations of pattern-centric graph mining for reducing both explicit and implicit redundancies.
- It proposes a dataflow-guided execution model and a self-managed redundancy storage substrate to efficiently share results among computations, overcoming the challenges of managing and reusing the results.
- It develops Cyclosa, a high-performance graph pattern mining system that eliminates redundant computations for various pattern settings. Experimental results show that Cyclosa outperforms GraphPi and SumPA by up to 16.28× and 5.52×, respectively.

2 Background and Motivation

2.1 Definition of Graph Pattern Mining

Given a data graph $G = \langle V, E \rangle$, where V is the vertex set and E represents the edges, and an input graph pattern p which can be arbitrary graphs, the basic graph pattern mining problem aims to find all the subgraphs of G that are isomorphic to p [20–22]. Each isomorphic subgraph is called an *embedding* of p, and the vertices and edges form an one-to-one mapping between the embedding and p. For example, Figure 1 shows a data graph and a diamond pattern. The subgraphs connected by (2, 3, 0, 4) and (2, 3, 1, 4) are two embeddings of the diamond pattern, and the vertices are mapped correspondingly. As in prior studies, this work focuses on graph pattern mining on the undirected graphs.

Graph pattern mining applications have many variants which may require either a single pattern or multiple patterns to be mined. For example, *subgraph listing* outputs all the instances of a given pattern [24]. The *k-clique finding* counts the complete subgraphs with a certain number of vertices [25]. The *k-motifs counting* counts the number of instances for all

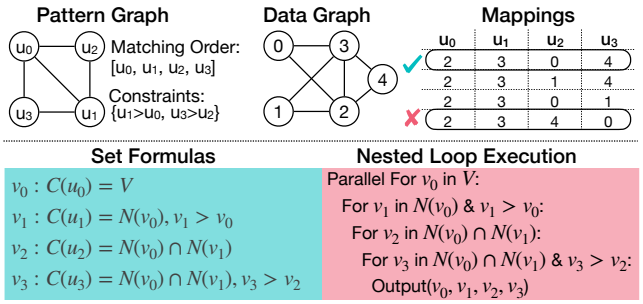


Figure 1: The example of mining a diamond pattern in a pattern-centric system

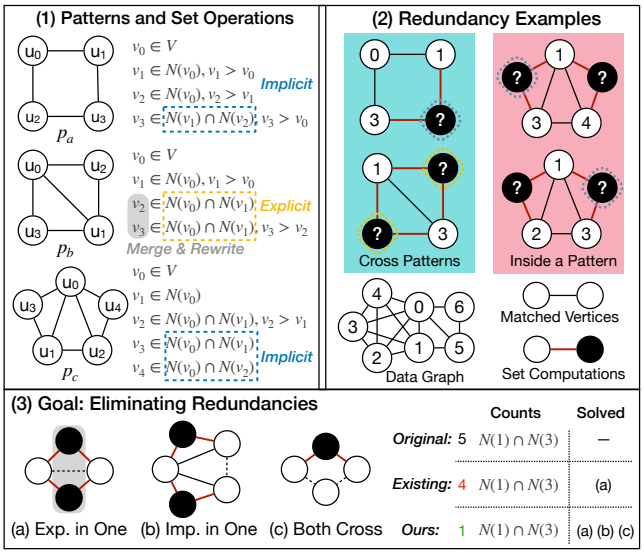


Figure 2: An example of redundant computations in graph pattern mining. The black vertices in dashed cycles represent redundant computations $N(1) \cap N(3)$ in different situations.

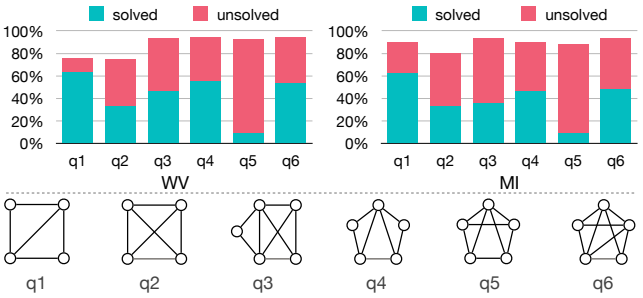


Figure 3: The ratio of redundancy as repeated set intersections for different patterns on *Wiki-Vote* (WV) and *MiCo* (MI) graphs. Solved: the redundancy amount that can be addressed by SumPA against a naïve nested loop execution. Unsolved: the number of remaining redundancies in SumPA.

2.2 Procedure of Graph Pattern Mining

In pattern-centric systems, the mining procedure is typically performed as a series of nested set operations in three steps [17, 22, 27]. Firstly, the graph pattern is analyzed to generate a matching order of pattern vertices. The vertices in the data graph will be computed and mapped to the pattern following the order. Secondly, the set operations required for computing each pattern vertex will be generated based on its prior neighbors in the matching order. Lastly, the set operations will be executed in a nested loop that starts from each data graph vertex. The end of each loop represents that a subgraph instance is found. Usually, the outer loop is executed in parallel. Existing systems focus on minimizing the branches that need to be accessed in the search tree by generating optimized matching orders. In order to further reduce the search space, the symmetry-breaking method is also applied by adding comparison constraints to filter computed results that will lead to an automorphic instance.

For instance, Figure 1 shows how to find the subgraphs of a diamond pattern. In the pattern analysis phase, the matching order of the diamond is defined as $[u_0, u_1, u_2, u_3]$. There are comparison constraints between vertex pairs $<u_1, u_0>$ and $<u_2, u_3>$ because they are symmetric. Following the matching order, we can formulate the set operations for each pattern vertex. Initially, u_0 can be mapped to any of the data graph vertices while u_1 is represented as finding a neighbor of u_0. For vertices u_2 and u_3, they are common neighbors of u_0 and u_1, so that a set intersection is defined respectively. The constraints are checked when $<u_1, u_0>$ and $<u_2, u_3>$ are involved in the computations. These set operations are organized into a nested loop of 4 depths for execution. The first two loops map edges of the data graph to (u_0, u_1). Assuming that the edge (2, 3) has been assigned, the candidates for u_2 and u_3 will be {0, 1, 4}. Due to the symmetry-breaking constraints, we can safely filter (2, 3, 4, 0) and avoid mapping (2, 3, 0, 4) twice because they are the same subgraph.

Our work incorporates existing optimizations of reducing the size of explored search space while enabling high efficiency on computations by eliminating redundancies.

2.3 Problem: Redundant Computations

Despite the pattern-centric mining procedure providing good optimizations on the search space, time-consuming explicit and implicit redundancies exist in the procedure of mining single or multiple patterns. This problem is detailed in Figure 2. The set operations for each vertex corresponding to the matching order and constraints of patterns P_a, P_b, and P_c are given. Following the matching order, consider that a part of

the substructures of these patterns have been mapped to the data graph, which is denoted as white vertices. Next, we will explain the explicit and implicit redundancies in detail.

Explicit Redundancy: A set of intersections that are repeatedly operated upon the neighboring lists of the same vertices from the *same* subgraph instance. For pattern P_b, (1, 3) is partially assigned to (u_0, u_1). In order to compute u_2 and u_3, $N(1) \cap N(3)$ is performed twice. For pattern P_c, the computation $N(1) \cap N(3)$ must be performed in order to map (1, 3, 4) to u_2. To this end, the explicit redundant computation $N(1) \cap N(3)$ is conducted based on the same edge instance (1, 3) inside single pattern P_b and also in the pattern P_c.

Implicit Redundancy: A set of intersections that are repeatedly operated upon the neighboring lists of the same vertices from the *different* subgraph instances. In single pattern P_c, consider two subgraph instances (1, 3, 4) and (1, 2, 3) are already mapped to (u_0, u_1, u_2). The repeated set intersections $N(1) \cap N(3)$ exist for computing u_3 and u_4. They are induced from different subgraph instances. For multiple patterns, when (0, 1, 3) is partially matched to P_a, the computation of $N(1) \cap N(3)$ is induced from different subgraph instances of all three patterns.

These redundancies can be aggravated in a nested loop of set operations as in Figure 1, e.g., resulting in more than twice the number of redundancies for computing u_3 of P_b. As profiled in Figure 3, more than 80% of total computations are redundant in different patterns. However, existing systems can only explore parts of the explicit redundancies because they view each set formula (or a pattern vertex) as a whole in each loop and all follow the principle of structural equality to merge equal set formulas. For example, in Figure 4, GraphPi [22] and GraphZero [21] will merge and rewrite the formulas S_3 and S_4 into one loop. SumPA [23] will reduce v_2 and v_3 into an abstract pattern vertex and then generate a single set formula for two vertices. However, the implicit redundancies remain because they cannot be exposed as structural equality in set formulas. Besides, in parallel execution, explicit redundancies on a reverted edge such as $N(1) \cap N(3)$ and $N(3) \cap N(1)$ will be omitted. As profiled in Figure 3, existing work can only solve less than 40% redundancies. Our work aims to eliminate both explicit and implicit redundancies.

2.4 Insights: Computation Similarity

In order to detect and reuse both the explicit and implicit redundancies, we must explore finer-grained computation similarity rather than the structural equality as in existing work. If the similarity of computations can be identified before the execution, then we can use it to guide the reusing of results. The computation similarity comes from two folds:

- *Static Similarity.* The static similarity originates from the operands level of the set operations for a pattern, exposing the reuse possibility of both inputs and outputs of different computations. Figure 4 presents the set formulas of P_c in Figure 2, other than the formula equality of S_3 and S_4, the input operand $N(v_0)$ is also getting involved in S_2 and S_5. $N(v_0)$ can be shared as inputs for four formulas, and it also results in the implicit redundancies between S_4 and S_5.

- *Dynamic Similarity.* The dynamic similarity lies in the runtime characteristics of the inputs of occurred computations, reflecting which vertices are more likely to be requested for computation. Specifically, we observe *most of the computations are concentrated in a small part of high-degree vertices.* We make an analysis of 4-motifs counting on two graphs as shown in Figure 5. The vertices of the graphs are reordered by a decreasing degree. More than 85% of the computations lie in about 15% of the first high-degree vertices, where the redundancies also concentrate.

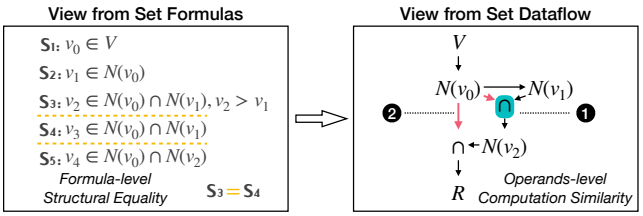


Figure 4: The dataflow view for analyzing the mining procedure of p_c in Figure 2. ❶ S_3 and S_4 are reduced. ❷ Results of $N(v_0)$ are reused in two ∩.

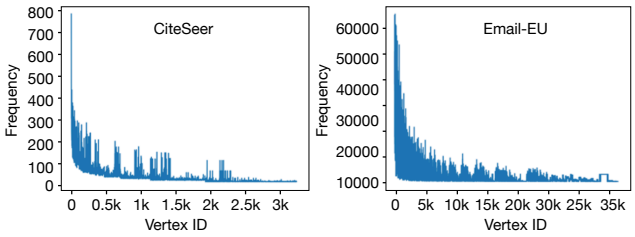


Figure 5: The frequency of vertices getting involved in computations for size-4 motifs counting on two graphs

In this work, we propose a *set dataflow* to use the computation similarity for redundancy elimination, as shown in Figure 4. The set dataflow is a directed graph indicating the procedure of how sets are transferred and computed. The set dataflow decouples the set formulas into individual operands and operators, and the directed edges represent the transfer relation of the input/output data between different operators. It removes explicit and implicit redundancies as follows: ❶ The explicit redundancies can be removed by cutting and maintaining unique operators, e.g., only single $N(v_0)$ and $N(v_1)$ exist. Original two $N(v_0) \cap N(v_1)$ are thus reduced to one, and the set operands are fully shared. ❷ The implicit redundancies between operators are indicated by overlapped inputs, e.g., the results of $N(v_0) \cap N(v_1)$ and $N(v_0) \cap N(v_2)$. Based on the dynamic similarity, we can heuristically cache the computation results of high-degree vertices for reusing.

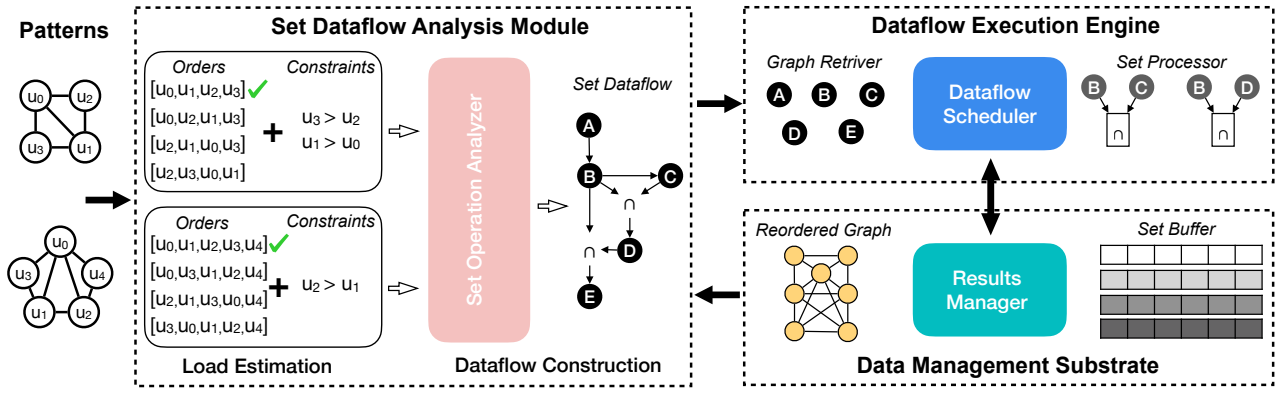


Figure 6: Overview of Cyclosa

Challenges. However, there are still several challenges in constructing an efficient redundancy-free graph pattern mining system. First, maintaining correctness and efficiency is difficult. The set dataflow is only aware of sets during execution, and we need to extract the results correctly while enabling maximum data sharing. Second, managing a large number of redundant results is challenging. Simply storing all the redundant results indicated by set dataflow is inefficient and may consume a large size of memory footprint.

3 System Overview

Cyclosa is architected to solve the above challenges and achieves a redundancy-free execution for graph pattern mining. Given arbitrary graph patterns, Cyclosa can fast analyze the computation similarity via the set dataflow and use the set dataflow to guide the sharing of computations with high parallelisms. Specifically, Cyclosa works with three main modules, as presented in Figure 6.

Set Dataflow Analysis Module. The set dataflow of input patterns is constructed in this module. Each pattern is first analyzed to generate a reuse-aware matching order and constraints with data graph properties. Then, based on the matching order, a set operation analyzer generates a redundancy-reduced set dataflow by keeping unique operands. Through the analysis, the generated set dataflow will maintain sufficient information for correct results and indicate the redundancy probability of different set operators.

Dataflow Execution Engine. The generated set dataflow is fed to the dataflow execution engine for processing. It keeps a decoupled view of the data access and computation that provides the opportunity to manage the results independently. Each node in the dataflow is assigned to a worker to process. The inputs and outputs are managed by the graph retriever communicating with the data management substrate. The set operators are assigned to redundancy-aware set processors that will request redundant results before the computation. A dataflow scheduler controls the processing order by directed edges of the set dataflow to ensure correctness.

Data Management Substrate. Computed results are automatically maintained in the substrate. Once received a result set, a results manager will implement smart cache strategies and maintain results with different reuse possibilities in a proper place of a multi-level set buffer. In this way, the memory footprint is under control. Furthermore, the results management can overlap with the execution engine for high parallelism. The substrate also provides a fast request to the data graph and cached results through efficient data layout.

4 Set Dataflow Analysis

This section first introduces the approach to generate a cost-efficient reuse-aware matching order and then describes the procedure for efficiently constructing a set dataflow.

4.1 Pattern Analysis

Existing approaches enumerate all possible orders and estimate the cost to find the order with the lowest workload [17, 20, 22], but they are oblivious to the redundant computations exposed in runtime characteristics. Besides, the overhead for enumerating all orders will increase when patterns get larger. In this work, we propose a degree-guided two phases analysis, as shown in Figure 7, to solve the above challenges. The main idea is to match the high-degree vertices in a *Depth-First-Search* (DFS) manner to raise the possibility of reusing the results on these vertices. At the same time, we design a lightweight and efficient cost model with graph information for the optimal cost.

DFS Order Enumeration. This phase will generate all degree-first DFS orders of a pattern. Firstly, the constraints of symmetric vertices are generated using the permutation group theory as in GraphPi [22], independent of the order of vertices. In the example of Figure 7, there are two equal constraints because each is sufficient to breaking symmetries, and the $u_0 < u_2$ is selected randomly. Secondly, it traverses

from the pattern vertices with the highest degree and follows a DFS style to get all valid matching orders, e.g., the order starting from u_1 with a degree by 4 is valid while the one from u_0 with a degree by 3 is discarded. These orders will then be estimated for the minimum cost.

Graph-Aware Cost Estimation. We estimate the total number of intermediate subgraph instances as the cost. Given a matching order, we can get the set formulas for computing each pattern vertex. Cyclosa then iteratively estimates the number of instances produced in a nested loop execution of the set formulas. Existing works typically use a fixed metric (i.e., average degree) to predicate the number of newly generated instances from each subgraph [17, 23]. However, this approach omits the filtering effect of set intersections and is inaccurate [28]. Cyclosa incorporates more graph information by combining the vertex degree and triangle-count-per-edge for estimation, because the triangle count enables capturing the reduction information on neighboring lists after an intersection on two or more vertices.

As shown in Figure 7, initially, $|V|$ vertices can be mapped to u_1 after `Loop0`. In `Loop1`, because only one $N(u_1)$ operator is used for matching u_0, we use the average degree deg to estimate the number of newly produced instances from each instance in the previous loop. The total instances in `Loop1` are $|V| * deg$. In `Loop2`, since the results are produced by an intersection on $N(u_0)$ and $N(u_1)$, we use the triangle-count-per-edge $ntri$, instead of deg, to estimate the number of newly generated instances. Note that the constraint $u_0 < u_2$ is applied to `Loop2` so that some produced instances will be filtered. We use a parameter α to reflect the reduction of instances. The cost estimated for subsequent loops is similar. Finally, the total cost of this order is calculated by summarizing the costs of all loops for selection.

Similar to Cyclosa, there is also research [28] considering the data graph properties and using the number of sub-patterns in a sampled graph to estimate the cost of the whole graph. However, it requires the extra sampling and matching phase to get the cost. Different from existing systems, Cyclosa precomputes triangle counts and degrees of the original graph to preserve accuracy without introducing extra steps. Additionally, Cyclosa also considers the factor of instances filtering by symmetry-breaking constraints.

4.2 Dataflow Construction

Given the matching order, a pattern can be transformed into a series of set formulas as in prior work. Based on the set operation, there are mainly two challenges for generating the set dataflow. First, the set dataflow must be aware of existing optimizations on symmetry-breaking. Second, we need a fast approach to save construction time when facing a large number of set operations.

We propose a novel abstraction for representing the set dataflow by introducing three kinds of set operators to provide

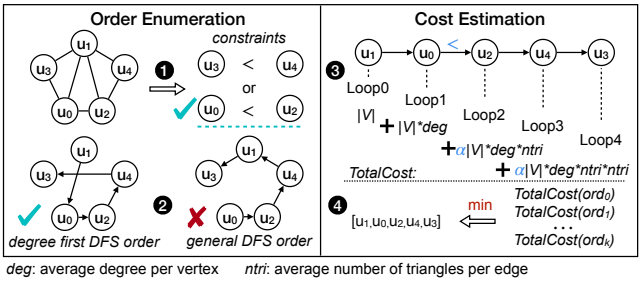


Figure 7: The procedure of finding an appropriate reuse-aware matching order for a pattern by ❶ generating constraints, ❷ enumerating valid DFS orders by degrees, ❸ estimating cost for each order with graph information, and ❹ selecting the order with minimum cost

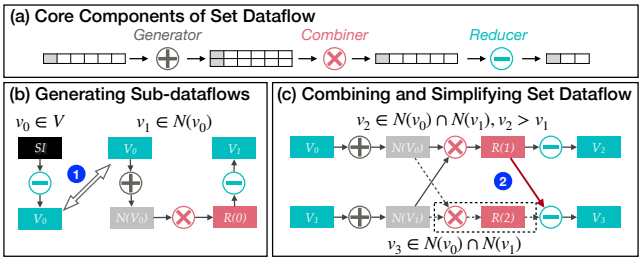
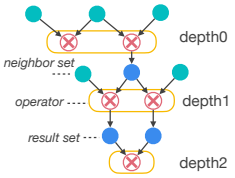


Figure 8: An example of constructing the set dataflow given four set formulas. The sub-dataflow of each set formula is first constructed with three operators, and the input/output sets of each operator are uniquely assigned and identified. The sub-dataflows are then combined into a final set dataflow by ❶ reducing inputs/outputs and ❷ simplifying set operators.

a uniform view and contain all information of set operations including the symmetry-breaking: `Generator`, `Combiner`, and `Reducer`. The `Generator` consumes a valid candidate set of a pattern vertex to generate the neighboring sets. The `Combiner` receives two sets and outputs a single set. The `Reducer` checks a result set and selects valid candidates following filtering rules to produce a new candidate set for certain pattern vertices. Based on these operators, we adopt the idea of divide-and-conquer by first generating the sub-dataflow of each set operation and then combining them to get the final set dataflow, as shown in Figure 8.

Generating Sub-Dataflows. The sub-dataflow of each formula must contain the `Generator`, `Combiner`, and `Reducer` at the same time, except of the initial one because there is no intersection required. In this way, the data required for computation and the operators performed are separated, which are represented in a unified style. For example, in Figure 8, the sub-flow of initial $v_0 \in V$ only contains one `Reducer`. For $v_1 \in N(v_0)$, the $N(v_0)$ is first transformed into a flow with a `Generator`, the output of the `Generator` is then sent to a `Combiner` and is finally checked by a `Reducer`. Similarly, the sub-dataflow for $v_2 \in N(v_0) \cap N(v_1)$ is given, and the con-



Probability. Reusable Results

	NST	HST	RST
Same-Depth	*high*	*middle*	*low*
Diff.-Depth	*middle*	*low*	*none*

Probability Table

	R(0)	R(1)	R(2)
R(0)	1	0.3	0.9
R(1)	0.3	1	0.3
R(2)	0.9	0.3	1

Figure 9: Different levels of redundancy probability. NST, HST, and RST denote operators sharing the same neighboring sets, sharing the same result set and different neighboring sets, and sharing only same result sets, respectively. The redundancy probability of each pair of result sets $(R(i), R(j))$ is stored in a table where higher values denote higher probability to be reused in the future.

straints will be recorded in the metadata of the `Reducer`.

Combining and Simplifying Set Dataflow. Different sub-flows are first combined by connecting same inputs and outputs, e.g., the sub-flows of $v_1 \in N(v_0)$ and initial formula are combined by the output and input candidate sets denoted with V_0. Only one candidate set of V_0 will be maintained after the combination. After the combination, we then traverse the combined set dataflow in a BFS style to cut set operators with the same input nodes. Consider two `Combiners` used for computing the results of v_2 and v_3. They have the same input nodes so that only one `Combiner` remains. Thus, the explicit redundancies of these two operators are eliminated. After the simplification, the final set dataflow is generated.

4.3 Dataflow Evaluation

This evaluation aims to exploit the latent probability of redundant results among different *Combiners* in the set dataflow, which will guide the storing of results. However, providing an exact prediction of the probability before execution is difficult. Therefore, we propose to qualitatively analyze the latent probability of redundancy for different set operators through their depths and input information in the dataflow.

Generally, the operators with at least one shared input source and at the same depth have a higher possibility of producing similar results. Considering the view of pattern structure, this can be understood as they have similar substructures. In Figure 2, the computations for u_3 and u_4 have produced the same results during runtime because these two vertices have similar triangle structures and share the u_0. Figure 9 summarizes the reuse probability under different situations of the combinations of depth and shared input sets. The redundant probabilities of different `Combiners` are recorded in a table together with the set dataflow for execution.

5 Redundancy-Free Pattern Mining

This section introduces an efficient set-centric execution engine and a redundancy-aware data management substrate to enable high performance and optimal results reuse.

5.1 Set-Centric Dataflow Execution

The set formulas are processed in a nested loop in prior works. Despite of the convenience of parallelizing, it lacks the ability of fine-grained data management and reuse. In this work, we present a set-centric dataflow execution engine that decouples data management from computation to maximize results sharing. The core idea is to put the set instead of a subgraph as the basic processed element. The set operator in the set dataflow will start processing whenever the input sets from directed edges are ready. This discrete view enables sharing any results to any of the computations.

To realize the goal, we correspondingly design an executor for each `Generator`, `Combiner`, and `Reducer` together with a *Dataflow Monitor*, as shown in Figure 10(a). Each set contains two parts for identification: the set ID and the elements value, e.g., ID $< 3, -1 >$ for the neighboring set of vertex 3. Set operators coordinate through flow signals. The monitor identifies from the flow signals to know where to move the results and activates the next operator.

Generator Module. It traverses each vertex element from the input candidate set or the initial set to generate related neighboring sets. Each input set is assigned a signal consisting of an operator ID and an instance ID. The output will inherit the operator ID, and a new instance ID is produced to indicate a newly generated subgraph instance for correctness. In the case of Figure 10(a), the signal $nei < 3, -1, op, gid^*, value >$ is sent to the monitor. The monitor will then transfer the output set to a `Combiner` needed.

Combiner Module. In order to support reuse-aware computation, it introduces a *check unit* and a *compute unit*. The check unit first queries whether there are already computed results with the same ID, $req < 3, 5, op, gid >$ in Figure 10(a). The monitor will transfer the request to the data manager. If the request hits, then the computation is omitted. Otherwise, the compute unit is called to generate a new output. The new result $res < 3, 5, gid, value >$ is then transferred to the monitor for the next operator.

Reducer Module. Each input to this module will be a result directly computed from the `Combiner` or fetched from the cached results. The output is a set with valid candidates for a pattern vertex. The constraint information is checked by accessing the metadata of the set dataflow. Once an output set is generated, the related signal, i.e., $ret < op, gid, value >$, is sent to the monitor.

Dataflow Monitor. The monitor coordinates the movement of sets guided by the set dataflow. It has three components: 1) the *Flow Map* storing the metadata of a set dataflow by recording unique IDs for operators, 2) the *Activator* handling the signals of operators and scheduling the sets based on the flow map, and 3) the *Retriever* communicating with the data manager for accessing the graph data and cached results.

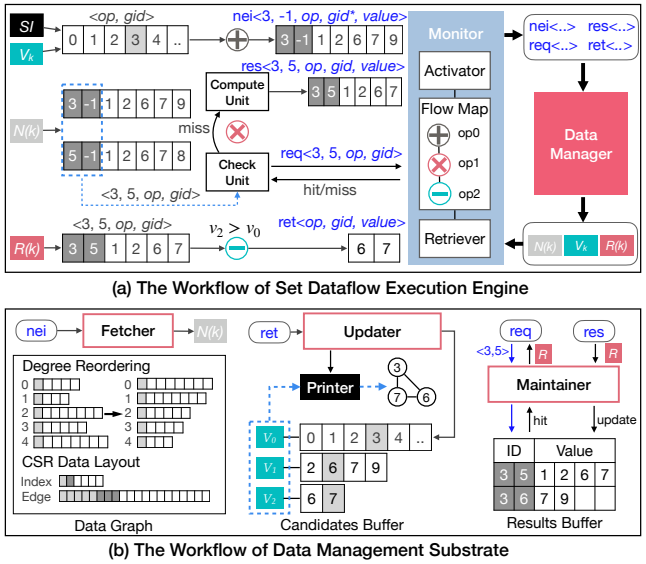


Figure 10: The workflows of set dataflow execution engine and data management substrate

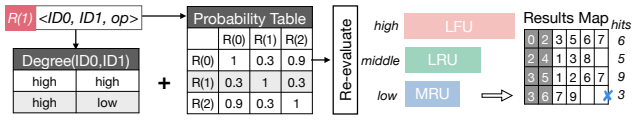


Figure 11: The caching strategy for result sets. The vertex degrees of result $R(1)$ are first checked, and only results with at least one high-degree vertex are maintained. The corresponding row of the probability table is then accessed to further decide the caching strategy.

5.2 Reuse-Aware Data Management Substrate

The dataflow execution engine will continuously produce new result sets. However, due to the large search space, maintaining the results in a limited memory space while maintaining high reuse ratio is challenging. We propose a memory-friendly data management substrate to solve this challenge. The main idea is to selectively store result sets with the highest reuse probability. It also overlaps with the computation phase to embrace correctness and efficiency. The substrate implements three main components: the results maintainer, the candidates updater, and the graph fetcher, as shown in Figure 10(b).

Results Maintainer. The cores of the results maintainer are three fixed-size result buffers under different caching strategies, i.e., the *Least Frequently Used* (LFU), the *Least Recently Used* (LRU), and the *Most Recently Used* (MRU) buffers storing results with high, middle, and low reuse probability, respectively. High reusable results come from high-degree vertices, which are more likely to be generated at the upper levels of the dataflow and are the most frequently requested. Middle reusable results are from parallel computations at the same middle level, yielding better time locality. Low reusable results occurs at the lower levels, which are often infrequently requested by low-degree vertices that are deferred to be processed in Cyclosa. This module dynamically maintains the computed results and responds to the *req* and *res* signals. The sets in the buffer are stored in a <key, value> manner. When a *req* signal arrives, it will search for results by the set ID, update the hit information, and respond to the dataflow monitor. When a *res* signal arrives, the maintainer will estimate the reuse probability with smart caching strategies and store the result in a proper buffer.

The strategy for identifying the reuse probability of a result set is demonstrated in Figure 11. It combines the static and dynamic computation similarities to estimate the reuse probability during runtime. Higher values in the probability table of the set dataflow analysis and higher degrees of the vertices by the result set ID, in particular, will result in a higher reuse probability evaluated. For instance, in Figure 11, the result has a low reuse probability. This is because there is a low degree vertex in the computation, which may not participate in other computations.

Candidates Updater. It is responsible for maintaining the candidate sets produced by the `Reducer` and extracting correct subgraph instances. Each candidate set is allocated with independent memory space. When a *ret* arrives, the *op* information is used for indicating the correct candidate sets. After each update, a *Printer* will consume the candidate sets using the *gid* information once all candidate sets are updated. The *Printer* records current pointers in the candidates set and produces exact subgraph instances. In order to overlap with the updating process, we use a double buffer for extracting the subgraph instances in the *Printer*.

Data Graph Fetcher. It reorganizes the data graph to improve the reuse efficiency and responds to the *nei* signal by returning the neighboring list as a set to the execution monitor. Specifically, based on the degree information, the vertices and edges are reordered so that the vertices with higher degrees will be assigned smaller IDs. In addition, the edges of the high-degree vertices are stored in a contiguous space to improve the cache efficiency because these edges will be frequently accessed during computation. It also conducts a triangle counting process to get the number of triangles of the data graph for pattern analysis.

Discussions. Through the above designs, Cyclosa enables efficient results sharing for redundant computations. In the case of operating intersections upon small sets, directly recomputing may be faster than reusing the results. However, this case rarely happens since most cached results are related to high-degree vertices that have large-size sets. We track the cached intersection results of size-4 motif counting on MiCo graph in Cyclosa and find that only 7.3% of results benefit from recomputations while most prefer the reuse method that can yield higher speedups against the former.

Currently, Cyclosa focuses on mining unlabeled patterns because they generally yield higher computation complex-

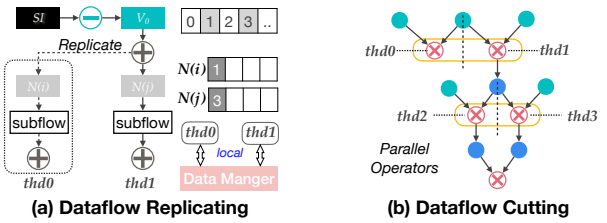


Figure 12: Parallel processing of the set dataflow that exploits multi-level parallelisms

Table 1: Real-World Graphs

Graphs	|V|	|E|	Size
WikiVote (WV)	7.1K	100.8K	0.81MB
MiCo (MI)	96.6K	1.1M	8.24MB
WikiTalk (WT)	2.39M	5.02M	40.16MB
Patents (PA)	3.8M	16.5M	0.12GB
LiveJournal (LJ)	4.0M	34.7M	0.26GB
Orkut (OR)	3.1M	117.2M	0.87GB
Friendster (FR)	65.6M	1.8B	13.46GB

ity than the labeled ones. For handling labeled patterns, we can simply add label information as constraints in `Reducers`. During execution, the structural information of a pattern is first explored for set computation, and the label information is then used only for filtering elements in a set. The labels provide more filtering opportunities that help reduce the total workload amount with fewer redundancies. However, the proposed set dataflow can still benefit labeled patterns by exploring the explicit redundancies in the pattern topology. Note that the dynamic computation similarity may change when the high-degree vertices have different labels.

6 Implementations

Cyclosa is currently built with C++ as an in-memory system on a single machine. This section will introduce the parallel implementations in Cyclosa for mining flexible patterns.

Flexible Pattern Interpretation. Cyclosa provides a convenient interface, `PatternGraph()`, for users to construct flexible pattern graphs by providing exact structures or graph properties, e.g., the edge list of a triangle or clustering coefficient, through *EdgeList* and *Restriction* parameters. The above information will be automatically interpreted into possible patterns. Users only need to interact with the `PatternGraph()` while the underlying runtime is transparent.

Parallel Execution of Set Dataflow. We use OpenMP for parallel execution in Cyclosa. The set dataflow inherently provides multi-level parallelisms, as shown in Figure 12. First, the sub-parts of a dataflow can be replicated and assigned to different threads to exploit the data parallelism. The starting point of the replicas is divided by the `Generator`. Take Figure 12(a) for example. Two threads handle the neighboring sets of vertex 1 and 3, respectively. Second, the `Combiners` at the same depth can be conducted in parallel because there are no dependencies, as shown in Figure 12(b). Note that the thread-local memory maintains an input set and an output set for every dataflow node which are reused throughout the execution. Since the number of dataflow nodes is small and the input/output set size is limited to the maximum degree of the data graph, the thread-local memory is often small.

Load Balancing. The skewness in Cyclosa relates to different numbers of sets produced by `Generators`. We address it in two folds: 1) *Static task assignment.* The input of the first `Generator` is initialized by assigning the reordered graph vertices in a round-robin fashion. This makes the upper `Generators` in a dataflow for different threads produce similar workloads. 2) *Dynamic work stealing.* This can be safely realized by managing the independent thread-local set space. Specifically, Cyclosa identifies the input-set position in the dataflow of the busy thread and replicates its workloads and relevant local set states to idle threads, which launch their corresponding `Generators` with higher parallelism.

Parallel Data Management. The data management substrate maintains the data graph and computation results. The data graph is stored in the *Compressed Sparse Row* (CSR) format. For cliques, the graph is oriented by enforcing a direction between each pair of vertices to reduce workloads. The results buffers are implemented using a concurrent hash map with a fixed-size space. Each candidate set is independently allocated with the size of the maximum degree. In this way, we can keep a controlled memory consumption.

7 Evaluation

In this section, we evaluate the effectiveness of the set dataflow and present the efficiency of Cyclosa.

7.1 Methodology

Patterns and Graph Datasets. The real-world graph datasets in our experiments are shown in Table 1, which are from the Stanford SNAP collection of datasets [29]. They represent typical graphs from different domains and are widely used in previous works [21–23]. The graph pattern mining algorithms evaluated are classified into two categories: 1) single pattern query that includes mining the single non-clique patterns (SM) [24] and *cliques finding* (CF) [25], 2) multiple patterns query that includes mining all patterns with a certain number of vertices, i.e., counting *k-motifs* (*k*-MC) [30] and multiple patterns satisfying specific graph property like *pseudo cliques* (PC) which are constrained with the given density [31]. These applications cover different kinds of representatives of graph pattern mining algorithms in prior works [20–23, 32].

Baseline Systems. Cyclosa is compared with two state-of-the-art pattern-centric systems, GraphPi [22] and SumPA [23].

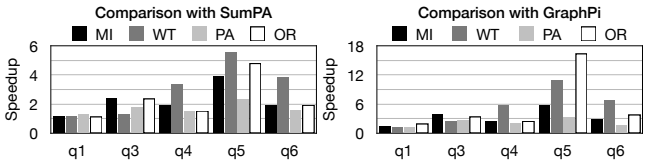


Figure 13: Performance comparison on listing single patterns

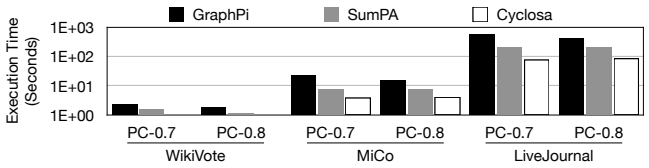


Figure 14: Performance on finding pseudo cliques

Table 2: Execution Time (Seconds) of Clique Finding

Systems	App.	MiCo	Patents	LiveJournal
Cyclosa	4-CF	0.23	0.11	2.27
	5-CF	21.82	0.29	463.02
SumPA	4-CF	1.21	0.37	5.98
	5-CF	50.77	0.45	1182.33
GraphPi	4-CF	1.64	0.44	12.14
	5-CF	60.51	0.52	1625.47

Table 3: Execution Time (Seconds) of Motifs Counting

Systems	App.	WikiVote	MiCo	Patents
Cyclosa	4-MC	1.25	9.56	6.81
	5-MC	257.34	1211.7	336.57
SumPA	4-MC	1.67	14.18	9.34
	5-MC	492.31	4789.5	662.03
GraphPi	4-MC	2.33	17.83	12.99
	5-MC	694.74	5803.97	803.55

Both systems are designed based on the mining procedure of Figure 1. GraphPi is a single pattern matching system that preserves the highest efficiency by finding an optimal combination of matching orders and symmetry constraints for an arbitrary pattern. SumPA is most related to our work and achieves higher performance on multiple patterns than prior works through a novel pattern abstraction.

Hardware Environments. All the experiments are conducted on a single server which is equipped with two 14-core Intel Xeon E5-2680v4 processors, 256GB RAM, and 512GB SSD. It runs a 64-bit Ubuntu 18.04 with kernel 5.4. We use gcc 7.3.0 to compile the applications with optimization under -O3. We use all the physical cores, and hyper-threading is enabled when the threads number exceeds 28.

7.2 Performance Comparison

Single Pattern Query. We first compare the performance of matching single non-clique patterns in Figure 3. These patterns are widely used in prior works [20, 21, 23]. Figure 13 shows the normalized speedups. Compared with GraphPi, Cyclosa achieves a speedup from 1.19× to 16.28×. The lowest speedup is for mining $q1$. Cyclosa can not only eliminate the explicit redundancies for $q1$ but also provide an efficient data graph layout. The highest speedup comes from $q5$ on Orkut. There are more implicit redundancies in $q5$ that cannot be removed by GraphPi, which can be explored in Cyclosa. Compared with SumPA, Cyclosa achieves a speedup from 1.13× to 5.52×. The pattern abstraction of SumPA can only expose parts of the explicit redundancies, while the set dataflow and smart cache in Cyclosa provide more opportunities for handling both explicit and implicit redundancies.

Table 2 shows the execution time for counting size-4 and size-5 cliques on different graphs. Overall, Cyclosa outperforms GraphPi by up to 7.13× and SumPA by up to 5.26× (4-CF on MiCo). GraphPi performs the lowest in all cases because all vertices in a clique are symmetric to each other, and the optimal matching order is unique. SumPA has limited improvement over GraphPi since the pattern abstraction will select a large sub-clique and omit opportunities for data reusing inside. In Cyclosa, the set dataflow can fully exploit the operands level redundancy for clique vertices, and the total workloads are reduced by graph orientation.

Multi-Pattern Query. We compare the performance of Cyclosa with SumPA and GraphPi on motifs counting and pseudo cliques. To support multiple patterns, we add a merging phase in GraphPi as in Automine [17]. Table 3 compares the execution time of counting size-4 and size-5 motifs on different graphs. With a larger size, the number of patterns increases (6 in 4-MC and 21 in 5-MC). Cyclosa outperforms SumPA and GraphPi by up to 3.95× and 4.79× for 5-MC on MiCo, respectively. The average speedups of Cyclosa on 4-MC and 5-MC of all cases are 1.64× and 2.95×. Note that Cyclosa achieves higher speedup when the pattern size and number increase. Existing approaches based on structural equality will face too many divergent branches when processing in-equal parts of these patterns. The set dataflow execution in Cyclosa can explore the computation similarity of both equal and in-equal parts.

Pseudo cliques (PC-k) algorithm finds patterns with a density greater than k. Figure 14 shows the results for mining all pseudo cliques with vertices less than six [23]. Notice that Cyclosa is superior to GraphPi and SumPA by 4.01× $\sim$ 7.52× and 1.48× $\sim$ 2.63×, respectively. Pseudo cliques are denser patterns, where one vertex is usually involved in most computations for other vertices. The set dataflow of Cyclosa can capture this similarity and fully reuse these operands as described in Section 2.4. Besides, the data management substrate can efficiently share results among different patterns.

Note that different mining algorithms can benefit from Cyclosa since the redundancies are highly related to the pattern topology and are independent of algorithm types. For larger patterns, more static similarity opportunities can be exploited. Also, an increased number of total computations amplifies the

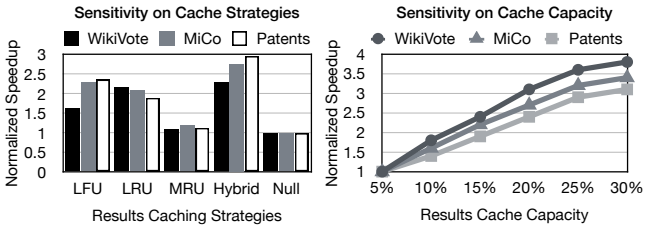


Figure 15: Normalized speedups under various settings of the cache capacity and caching strategy for 4-MC

Table 4: Execution Time (Seconds) on the Large Graph (FR)

App.	Cyclosa	SumPA	GraphPi
3-MC	94.26	143.38	281.42
4-CF	512.97	1491.72	1647.55

reusable computation amount arising from dynamic similarity. However, the size of set dataflow may increase and require more local memory space for parallel processing.

7.3 Sensitivity Study

Cache Strategy. The left chart in Figure 15 investigates the behaviors of different caching strategies for 4-MC. We fix the buffer capacity as 15% size of corresponding graphs. Notice that no single strategy can outperform others in all cases. Among all graphs, the LFU behaves better on Patents graph (2.36×), and the LRU behaves better on the WikiVote (2.17×). This is due to the diverse sparsity of the graphs, and the Patents graph is sparser than WikiVote. The MRU strategy is slowest in all cases, and this is because the results of high-degree vertices in the prior phase will be discarded even though they may be frequently reused. The hybrid strategy can combine the pattern and graph information to select the best buffer and thus achieves the highest performance.

Cache Capacity. The right chart in Figure 15 shows the normalized speedups with various buffer sizes. Initial buffer size is defined as the 5% size of a given data graph. Typically, larger cache capacity brings higher performance gains, e.g., 3.1× improvement from 5% to 20% on WikiVote, because more results can be cached and reused. However, the growth slows down gradually while increasing the buffer size. The main reason is that a larger buffer size induces higher latency in maintaining and querying the results.

7.4 Scalability

Figure 16 compares the performance of different systems by varying the number of threads. The results are normalized to GraphPi with a single thread. Hyperthreading is enabled when the number of threads exceeds 28. For GraphPi and SumPA, each thread is saturated with computations. More threads offer more compute parallelism but have to compete for computation resources. Thus, hyperthreading improves efficiency,

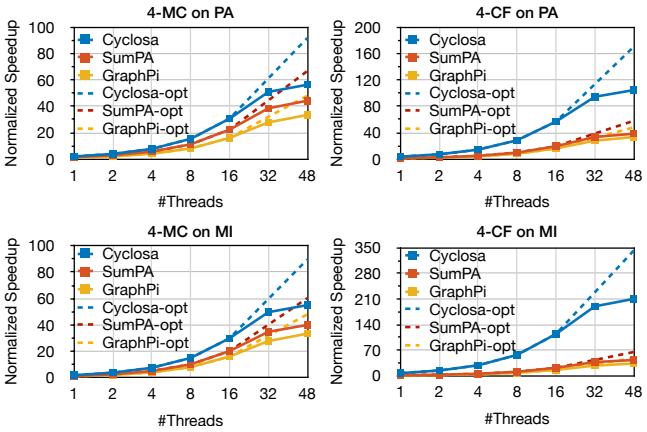


Figure 16: The normalized speedups with various number of threads for different applications

Table 5: Memory Consumption on Orkut with 28 Threads

Systems	4-CF	4-MC	5-MC	PC-0.8
Cyclosa	2.69GB	2.75GB	2.96GB	2.94GB
GraphPi	3.81GB	3.83GB	4.16GB	3.97GB

but the growth slows down gradually. Cyclosa resolves the redundant computation bottleneck, therefore offering more speedups against the above earlier systems. However, memory access contention becomes important when hyperthreading is used, incurring the increasingly-saturated performance improvement as shown in Figure 16. This is because Cyclosa has to maintain and query the results cache.

We also test the ability of Cyclosa to scale to large graphs with billion edges, as shown in Table 4. For size-3 motifs counting, Cyclosa gains 1.52× and 2.99× speedups over SumPA and GraphPi. Cyclosa outperforms SumPA and GraphPi on the Friendster graph for size-4 clique by 2.91× and 3.21×, respectively. Caching the results for large graphs is difficult because of the vast amount of intermediates. The performance improvement proves the efficiency of the set dataflow and caching strategies on larger graphs.

7.5 Overhead

Memory Consumption. Table 5 compares the memory consumption of Cyclosa and GraphPi on the Orkut graph. Cyclosa and GraphPi take an average of 2.84GB and 3.94GB of memory space in these cases. Although Cyclosa needs to maintain some of the results, the memory footprint is still kept small. This is because GraphPi maintains intermediate subgraph instances while Cyclosa shares the same result set for different vertices and executes the dataflow in a DFS style. Each thread in Cyclosa only maintains a local space for each set dataflow node. Besides, the smart caching strategies explore the trade-off between the space efficiency and reuse possibility. The memory space in the data management substrate is

Table 6: Time for Constructing Set Dataflow

App.	WikiVote	Patents	Orkut
5-CF	1.74ms	1.72ms	1.69ms
5-MC	9.32ms	9.44ms	9.37ms
PC-0.8	3.16ms	3.09ms	3.22ms

pre-allocated by a small fixed size.

Dataflow Construction. Table 6 presents the dataflow constructing time for different applications. This includes pattern analysis to get matching order, building the set dataflow, and evaluating redundancy probability. The triangle counting time is excluded because it is only conducted once and can be reused for different algorithms. Notice that the constructing time rises when the number of patterns increases. However, the dataflow construction is only executed once throughout processing and is negligible compared with the computation time. For example, it takes over 250s for 5-MC on WikiVote, while the dataflow construction only takes 9.32ms.

8 Related Work

Early graph pattern mining research focuses on customized improvements for specific given algorithms. For counting cliques, kClist [33] designs an efficient algorithm for processing sparse graphs based on the core value. To handle motifs effectively, G-tries [34] creates a novel tree-like data structure. PGD [26] counts all size-3 and size-4 motifs using partial patterns by some combinatorial rules. There have also been works that use GPUs to speed the subgraph isomorphism problem in finding network motifs and enumerating subgraphs [18, 35–37]. Cyclosa, as opposed to algorithm-specific improvements, focuses on tackling the common redundancy challenges in a more general situation. Prior optimizations can also be integrated into Cyclosa.

General-purpose graph pattern mining systems use expressive and efficient programming paradigms to automatically parallelize a variety of graph mining applications in a consistent manner [16, 38]. Early distributed systems, such as Arabesque [16] and Fractal [19], adopt the embedding-centric model to iteratively extend and enumerate all subgraphs size by size and verify the user-defined constraints for each intermediate embedding. RStream [39] and Kaleido [40] optimize the embedding-centric model in an out-of-core manner on a single machine to alleviate the data shuffling and communication cost. Pangolin [41] and Sandslash [42] provide flexible interfaces that integrate customized algorithmic optimizations to enhance the filtering of intermediate embeddings. Despite the expressiveness and massive parallelisms of these systems, managing a large number of intermediate embeddings becomes the main bottleneck, which suffers from high memory consumption and heavy I/Os [15].

Compared to the embedding-centric model, Cyclosa works in a pattern-aware manner, pioneered by AutoMine [17] and Peregrine [20], to avoid unnecessary storage and process of embeddings under the guide of pattern constraints. The cores of these systems are efficient matching engines that execute fast set operations [27]. AutoMine is a compilation-based system that automatically transforms graph patterns into set programs. GraphZero [21] is an enhanced version of AutoMine that removes explicit redundancies among multiple patterns by introducing the symmetry-breaking optimizations that define a partial order between symmetric vertices. GraphPi [22] further explores the optimal combination of matching orders and symmetry-breaking constraints to speed set programs. Despite the efficiency, the compilation method may induce non-negligible overhead while generating new set programs for newly coming patterns. Dryadic [43] proposes a flexible tree-structured intermediate representation to solve this problem, which supports both compilation and runtime optimizations. DecoMine [28] decomposes a large pattern into smaller ones for faster pattern counting. SumPA [23] merges multiple patterns in a pattern abstraction to minimize workloads. Cyclosa aims to eliminate the inherent computation redundancies, and the data management substrate can also be integrated into the above systems to reduce redundancies.

9 Conclusion

In this work, we present a redundancy-free framework, Cyclosa, that eliminates both explicit and implicit redundancies in pattern-centric graph mining systems. Cyclosa explores the computation similarity through a novel set dataflow representation, which exploits a finer-grained similarity at the operand level instead of the structural equality as in existing works, making it possible to indicate both explicit and implicit redundant computations. Based on the set dataflow, Cyclosa implements an efficient dataflow-guided execution model collaborated with a memory-friendly data management substrate to efficiently reuse computing results, embracing high performance and correctness. The proposed substrate may also be incorporated into the runtime of existing systems to reduce redundancies. Evaluation of a variety of patterns and real-world graphs shows that Cyclosa can significantly outperform state-of-the-art systems GraphPi by up to 16.28× and SumPA by up to 5.52× for various applications.

Acknowledgments

We thank our anonymous reviewers and shepherd for their insightful suggestions. This work is supported by the National Key Research and Development Program of China under (Grant No. 2022YFB4501403), the NSFC (Grant No. 61832006 and 61929103), the Zhejiang Lab (Grant No. 2022PI0AC02), and the Fundamental Research Funds for the Central Universities (Grant No. YCJJ202202011). Long Zheng is the corresponding author.

References

[1] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference*, pages 49–60, 2013.

[2] Vasileios Trigonakis, Jean-Pierre Lozi, Tomás 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 *Proceedings of the USENIX Annual Technical Conference*, pages 209–224, 2021.

[3] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018.

[4] Dan Chen, Chuangyi Gui, Yi Zhang, Hai Jin, Long Zheng, Yu Huang, and Xiaofei Liao. Graphfly: Efficient asynchronous streaming graphs processing via dependency-flow. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2022.

[5] Yuhang Wang, Fillia Makedon, James Ford, and Heng Huang. A bipartite graph matching framework for finding correspondences between structural elements in two proteins. In *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 2972–2975, 2004.

[6] Bibek Bhattarai, Hang Liu, and H. Howie Huang. CECI: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the ACM International Conference on Management of Data*, pages 1447–1462, 2019.

[7] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. RapidMatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment*, 14(2):176–188, 2020.

[8] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Parallel k-clique counting on GPUs. In *Proceedings of the ACM International Conference on Supercomputing*, pages 1–14, 2022.

[9] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. In *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms*, pages 135–146, 2021.

[10] Pedro Ribeiro, Pedro Paredes, Miguel E. P. Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys*, 54(2):28:1–28:36, 2021.

[11] Lorenzo De Stefani, Erisa Terolli, and Eli Upfal. Tiered sampling: An efficient method for counting sparse motifs in massive graph streams. *ACM Transactions on Knowledge Discovery from Data*, 15(5):79:1–79:52, 2021.

[12] Nishil Talati, Haojie Ye, Sanketh Vedula, Kuan-Yu Chen, Yuhan Chen, Daniel Liu, Yichao Yuan, David Blaauw, Alex Bronstein, Trevor Mudge, and Ronald Dreslinski. Mint: An accelerator for mining temporal motifs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 1270–1287, 2022.

[13] 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*, pages 716–727, 2016.

[14] 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 ACM European Conference on Computer Systems*, pages 1–12, 2018.

[15] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. A locality-aware energy-efficient accelerator for graph mining applications. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 895–907, 2020.

[16] 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 ACM Symposium on Operating Systems Principles*, pages 425–440, 2015.

[17] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 509–523, 2019.

[18] Xuhao Chen and Arvind. Efficient and scalable graph pattern mining on GPUs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 857–877, 2022.

[19] 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 ACM SIGMOD International Conference on Management of Data*, pages 1357–1374, 2019.

[20] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the ACM European Conference on Computer Systems*, pages 1–16, 2020.

[21] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. GraphZero: A high-performance subgraph matching system. *ACM SIGOPS Operating Systems Review*, 55(1):21–37, 2021.

[22] 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*, pages 1–14, 2020.

[23] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. SumPA: Efficient pattern-centric graph mining with pattern abstraction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 318–330, 2021.

[24] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. HUGE: An efficient and scalable subgraph enumeration system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2049–2062, 2021.

[25] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. Lightning fast and space efficient k-clique counting. In *Proceedings of the ACM Web Conference*, page 1191–1202, 2022.

[26] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *Proceedings of the IEEE International Conference on Data Mining*, pages 1–10, 2015.

[27] Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, Peter Tatkowski, Esref Ozdemir, Adrian Balla, Marcin Copik, Philipp Lindenberger, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. GraphMineSuite: Enabling high-performance and programmable graph mining algorithms with set algebra. *Proceedings of the VLDB Endowment*, 14(11):1922–1935, 2021.

[28] Jingji Chen and Xuehai Qian. DecoMine: A compilation-based graph pattern mining system with pattern decomposition. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 47–61, 2022.

[29] Jure Leskovec and Andrej Krevl. SNAP datasets: Stanford large network dataset collection. 2014. http://snap.stanford.edu/data.

[30] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. LINC: A motif counting algorithm for uncertain graphs. *Proceedings of the VLDB Endowment*, 13(2):155–168, 2019.

[31] Takeaki Uno. An efficient algorithm for enumerating pseudo cliques. In *Proceedings of the International Symposium on Algorithms and Computation*, pages 402–414, 2007.

[32] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, Wei-Shinn Ku, , and John C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1369–1380, 2020.

[33] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *Proceedings of the International World Wide Web Conference*, pages 589–598, 2018.

[34] Pedro Ribeiro and Fernando Silva. G-Tries: An efficient data structure for discovering network motifs. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1559–1566, 2010.

[35] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiaoli Li. Network motif discovery: A GPU approach. *IEEE Transactions on Knowledge and Data Engineering*, 29(3):513–528, 2017.

[36] Li Zeng, Lei Zou, M. Tamer Özsu, Lin Hu, and Fan Zhang. GSI: GPU-friendly subgraph isomorphism. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1249–1260, 2020.

[37] 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 ACM SIGMOD International Conference on Management of Data*, pages 1067–1082, 2020.

[38] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. Tesseract: Distributed, general graph pattern mining on evolving

graphs. In *Proceedings of the European Conference on Computer Systems*, pages 458–473, 2021.

[39] 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 USENIX Symposium on Operating Systems Design and Implementation*, pages 763–782, 2018.

[40] 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 IEEE International Conference on Data Engineering*, pages 673–684, 2020.

[41] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on CPU and GPU. *Proceedings of the VLDB Endowment*, 13(10):1190–1205, 2020.

[42] 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 ACM International Conference on Supercomputing*, pages 378–391, 2021.

[43] Daniel Mawhirter, Samuel Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and fast graph pattern matching at scale. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 289–303, 2021.

SOWalker: An I/O-Optimized Out-of-Core Graph Processing System for Second-Order Random Walks

Yutong Wu, Zhan Shi, Shicai Huang, Zhipeng Tian, Pengwei Zuo,
Peng Fang, Fang Wang, Dan Feng
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Corresponding Author: Zhan Shi (zshi@hust.edu.cn)

Abstract

Random walks serve as a powerful tool for extracting information that exists in a wide variety of real-world scenarios. Different from the traditional first-order random walk, the second-order random walk considers recent walk history in selecting the next stop, which facilitates to model higher-order structures in real-world data. To meet the scalability of random walks, researchers have developed many out-of-core graph processing systems based on a single machine. However, the main focus of out-of-core graph processing systems is to support first-order random walks, which no longer perform well for second-order random walks.

In this paper, we propose an I/O-optimized out-of-core graph processing system for second-order random walks, called SOWalker. First, we propose a walk matrix to avoid loading non-updatable walks and eliminate useless walk I/Os. Second, we develop a benefit-aware I/O model to load multiple blocks with the maximum accumulated updatable walks, so as to improve the I/O utilization. Finally, we adopt a block set-oriented walk updating scheme, which allows each walk to move as many steps as possible in the loaded block set, thus significantly boosting the walk updating rate. Compared with two state-of-the-art random walk systems, GraphWalker and GraSorw, SOWalker yields significant performance speedups (up to 10.2×).

1 Introduction

Random walks on graphs have received significant attention for their ability to extract meaningful insights in graph data analysis and machine learning [10–14]. Most existing random walk implementations are based on the first-order Markov model [15, 16], which assumes the transition probability only depends on the current vertex and is independent of the previous information. Although many encouraging results have been obtained under this assumption, the high-order information such as second-order properties is ignored, and thus some recent works have revealed the necessity of second-order random walks [17, 18]. Node2vec [14], one of the most popular network embedding methods, uses the second-order random walk to capture neighborhood information of vertices, which significantly outperforms the first-order methods like DeepWalk [13]. Similar findings have been found in graph proximity measurements. Wu et al. [19] developed the second-order PageRank and SimRank, and Liao et al. [20] proposed the second-order CoSimRank, which can explore cluster structures in the graph and better model real-world applications. In addition, the random walk has been widely used in social physics. Rosvall et al. [21] showed how the second-order random walk model constraints on dynamics influence community detection, ranking, and information spreading, while it is difficult for the first-order random walk in these scenarios.

Graphs with billions of edges are becoming more prevalent in many domains, and performing some tasks often requires tens of TB to several PB spaces. Although some vendors such as Amazon (AWS) [6], Oracle [7], and Microsoft [8] provide graph database services, striking a balance between low cost and high quality remains challenging. For example, when processing our largest graph, CrawlWeb (see Section 4.1), on Amazon Neptune [9], we use an Amazon Elastic Compute Cloud (EC2) instance, db.r5.4xlarge (8 cores, 16 virtual cores, 128 GB memory), and 3 TB of storage space. The monthly cost for this instance is up to $3, 000. As the size of the graph continues to grow, the storage requirements and computational resources needed would also increase. This would likely lead to higher costs in terms of storage space, computing resources, and potentially data transfer. In contrast, out-of-core graph processing systems are cheaper and easier, as they utilize external storage for processing large graphs [22–25]. These systems divide a large graph into several blocks (i.e., subgraphs) and store them on disks. During the graph processing, a block is loaded into memory and application-specific vertex or edge values in this block can be updated immediately. As expected, the significant performance bottleneck of out-of-core graph processing systems is the I/O between memory and disks, and developers can ill afford to ignore it. Recently, numerous works have been devoted to designing I/O-efficient graph processing systems for random walks.

DrunkardMob [26] is the first large-scale out-of-core graph processing system that allows massive random walks to be performed in parallel. The states of all walks are represented compactly in memory to minimize the memory footprint of each walk. GraphWalker [27] adopts a state-aware I/O model and an asynchronous walk updating scheme to further improve the I/O performance. Besides, it proposes a lightweight block-centric indexing scheme to reduce the memory requirement for storing walk states. However, these proposed solutions cannot be efficiently compatible with second-order random walks.

In this paper, we propose an I/O-optimized out-of-core graph processing system for second-order random walks, called SOWalker. First, to eliminate useless walk I/Os, we propose a *walk matrix* to represent the walks, so as to prevent loading non-updatable walks whose previous vertices are not in memory. Along with the proposed walk matrix, we design a succinct and compact data structure to provide an efficient representation of a walk. Second, to improve the I/O utilization, we develop a *benefit-aware I/O model*. Specifically, we load multiple blocks with the maximum accumulated updatable walks and only load walks whose previous and current vertices are both in the loaded blocks. For this purpose, we map the block scheduling problem into the maximum edge weight clique problem, and adopt a heuristic algorithm to provide comparable I/O performance but significantly reduce the computation time. Finally, to boost the walk updating rate, we adopt a *block set-oriented walk updating scheme*, which allows each walk can be updated as much as possible in the loaded block set, so as to accelerate the progress of random walks. To summarize, we make the following contributions.

- We propose a *walk matrix*, which prevents loading non-updatable walks, so as to eliminate useless walk I/Os.
- We develop a *benefit-aware I/O model*, which loads multiple blocks with the maximum accumulated updatable walks, so as to maximize the I/O utilization.
- We adopt a *block set-oriented walk updating scheme*, which allows each walk to move as many steps as possible in the loaded block set, so as to boost the walk updating rate.
- We conduct detailed experiments on a variety of real-world and synthetic graphs to evaluate SOWalker. Extensive evaluation results show that SOWalker can substantially reduce the I/O cost, achieving up to 10.2× speedup.

The rest of this paper is organized as follows. Section 2 presents the background and motivation. Section 3 describes the detailed system designs of SOWalker. Section 4 evaluates the system and compares it with two state-of-the-art systems. Section 5 gives an overview of related work, and finally, Section 6 concludes this paper.

2 Background and Motivation

Given a graph $G = (V, E)$, where V and E are the set of vertices and edges, respectively. Each edge $e \in E$ is an ordered pair $e = (u, v)$ and is associated with a weight w_{uv}. For each $u \in V$, the neighbor set of a vertex u is $N(u)$. For easy reference, we illustrate the frequently used notations in Table 1.

Notation	Description
$G = (V, E)$	graph G with vertex set V and edge set E
$e = (u, v)$	edge from u to v
w_{uv}	weight between vertex u and v
$N(u)$	set of neighbor vertices of vertex u
B	block set
$\|B\|$	number of blocks in B
m	maximum number of blocks cached in memory
B_L	loaded block set in a batch
$W(i, j)$	number of walks crossing between block i and j
AUW	accumulated updatable walks
CDG	complete directed graph
k	actual number of blocks to be loaded
β	a bitmap to represent whether the block is cached in memory
T_0, T_s	initial temperature and end temperature
γ	cooling coefficient of temperature
$iter_{max}$	maximum number of iterations

Table 1: Notations.

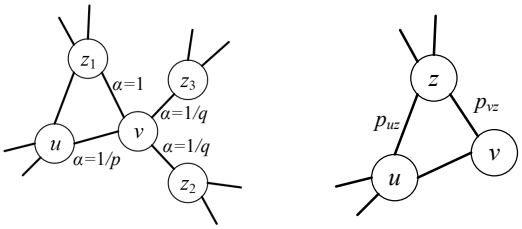
2.1 Second-order Random Walk

First-order random walk. Suppose a walk is visiting vertex v, in the next step, the walk will move to a neighbor of v with transition probability $p_{vz} = P(z|v) = w_{vz}/W_v$, where $W_v = \sum_{t \in N(v)} w_{vt}$.

Second-order random walk. Given that the walk is visiting vertex v at the current step and vertex u at the previous step, the second-order transition probability that moving to vertex z at the next step is $p_{uvz} = p(z|uv)$. Such transition probability can be interpreted as the edge-to-edge transition probability: let $\alpha = (u, v)$ be the edge from vertex u to v, and $\beta = (v, z)$ be the edge from vertex v to z, that is, $p_{uvz} = p_{\alpha\beta}$.

Below are two representative examples of second-order random walk-based algorithms.

Node2vec. Node2vec [14] is a popular network embedding method that introduces the second-order random walk. In order to combine Depth First Search (DFS) and Breadth First Search (BFS), two parameters p and q control the random walk strategy. Parameter p controls the probability of repeated access to the just visited vertex. Parameter q controls whether a walk moves inward or outward. Given that vertex u was visited at the previous step, the unnormalized transition probability p_{uvz} from the current vertex v to the



(a) Node2vec (b) Second-order PageRank

Figure 1: The transition probability computation in second-order random walk algorithms.

next vertex z depends on the edge weight w_{vz} and $\alpha_{pq}(u,v,z)$, i.e., $p_{uvz} = \alpha_{pq}(u,v,z) \cdot w_{vz}$. α_{pq} is calculated in the following formula (shown in Figure 1(a)):

$$\alpha_{pq}(u,v,z) = \begin{cases} \frac{1}{p}, & d_{uz} = 0 \\ 1, & d_{uz} = 1 \\ \frac{1}{q}, & d_{uz} = 2 \end{cases}$$

where d_{uz} denotes the shortest path distance between vertices u and z, and $d_{uz} \in \{0,1,2\}$.

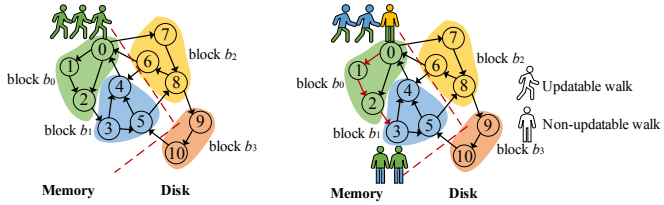
Second-order PageRank. Wu et al. [19] proposed a second-order PageRank and used an autoregressive model to compute the second-order influence probability, which is described as follows:

$$p_{uvz} = \frac{p'_{uvz}}{\sum_{t \in N(v)} p'_{uvt}}$$

where $p'_{uvz} = (1-\alpha)p_{vz} + \alpha p_{uz}$ (shown in Figure 1(b)). The parameter $\alpha \in [0,1)$ is a constant (e.g., 0.2) to control the strength of effect from the previous step.

2.2 Motivation

The out-of-core random walk systems divide a graph into several blocks and cache some of them in memory, the remainder blocks reside in disks temporarily. The total number of cached blocks is limited by the available memory and block size. During the random walk procedure, a block is loaded from disk into memory according to a scheduling model, which is called the *current block*. Walks that reside on the current block are also loaded into memory and can be updated as much as possible until they reach the boundary of the loaded block. However, the main focus of out-of-core random walk systems is to support first-order random walks, which are no longer effective for second-order random walks. In the following, we will discuss three major challenges that lead to poor I/O performance on existing systems.



(a) First-order random walks (b) Second-order random walks

Figure 2: Differences in updatable walks between (a) first-order random walks and (b) second-order random walks. All three first-order random walks are updatable as opposed to only two second-order random walks. View in color for optimal visualization.

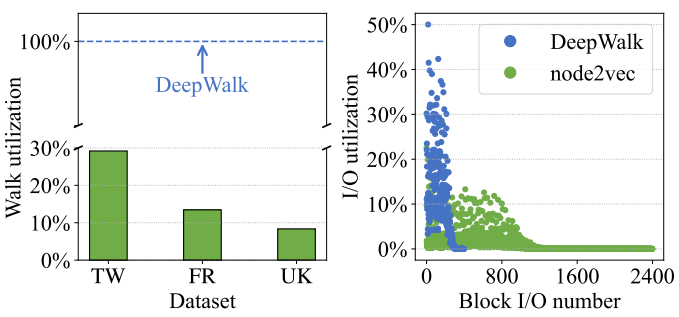
Non-updatable walks result in useless walk I/Os. The first-order random walk's transition probability only depends on the current vertex. As long as the current vertex is in memory, the walk can be immediately updated. Unlike the first-order random walk, the second-order random walk considers recent walk history in selecting the next stop. However, the previous vertex might belong to other blocks on slow disks. Consequently, due to the lack of previous vertex information, some loaded walks cannot be updated directly, resulting in *non-updatable walks*. As a result, these non-updatable walks lead to useless walk I/Os.

As an example, Figure 2 illustrates the differences in updatable walks between first-order and second-order random walks for a specific iteration, where block b_0 and b_1 are in memory, and block b_0 is the current block. Suppose that there are three walks residing on vertex 0. For first-order walks in Figure 2(a), all three walks are updatable. As a result, the walk utilization of first-order random walks is always 100%, which is defined as the ratio of updatable walks to the total loaded walks. For second-order walks in Figure 2(b), the color of the walk represents its state, with the upper and lower colors indicating the block that the previous and current vertex belongs to, respectively. Out of the three walks, one has its upper half-colored yellow, indicating that its previous vertex belongs to block b_2, which is not in memory. As a result, this walk is non-updatable, resulting in the walk utilization of only 2/3. In order to further quantitatively evaluate the walk utilization of node2vec on real-world graphs, we conducted experiments on three datasets (introduced in Section 4.1). As shown in Figure 3(a), the walk utilization of node2vec is less than 30%, and it decreases as the size of the graph increases.

In SOWalker, we propose a walk matrix, which prevents loading non-updatable walks to eliminate useless walk I/Os.

The non-optimal block scheduling model results in low I/O utilization. To update non-updatable walks, the existing block scheduling model [27, 28] iteratively loads ancillary blocks where previous vertices belong to, resulting in a large number of additional block I/Os. On the other hand, due to the irregular structure of graphs and the randomness inherent in random walks, previously visited vertices are unevenly scattered in different blocks.

To quantify the effect of the non-optimal block scheduling model on I/O utilization, we run DeepWalk (i.e., first-order) and node2vec (i.e., second-order) on a state-of-the-art system, GraphWalker [27]. The I/O utilization is defined as the num-



(a) Walk utilization (b) I/O utilization

Figure 3: Walk utilization and I/O utilization.

ber of walk steps divided by the number of edges in a loaded block. For Deepwalk, we use GraphWalker's state-aware I/O model to preferentially load the block with the most walks into memory. For node2vec, we also load a block with the most walks as the current block and iteratively load another block into memory as the ancillary block. Figure 3(b) shows the I/O utilization of DeepWalk and node2vec. We can see that the I/O utilization is significantly low in the second-order random walk application. Besides, running DeepWalk requires fewer than 400 block I/Os, while running node2vec requires over 2400 block I/O, which severely slows down the processing of random walks.

In SOWalker, we develop a benefit-aware I/O model, which loads multiple blocks with the maximum accumulated updatable walks to maximize the I/O utilization.

The block-oriented walk updating scheme brings low walk updating rate. Existing systems [27] manage walks at a block granularity and restrict walk updating to a block, which is called *block-oriented walk updating*. However, this hinders the walk updating and walks fail to utilize the vertex information in other blocks residing in memory, resulting in low walk updating rate. For example, in Figure 2(b), suppose that two updatable walks move along the red path toward vertex 3. Under the block-oriented walk updating scheme, block b_1, where vertex 3 belongs to, is not the current block, so the walks cannot continue to move, which leads to low walk updating rate. In fact, if a walk moves to any vertex belonging to the block in memory, it can further be updated, since the previous and current vertex information are both available.

In SOWalker, we adopt a block set-oriented walk updating scheme, which allows each walk to move as many steps as possible in the loaded block set to boost the walk updating rate.

3 Design of SOWalker

In this section, we first present the system overview of SOWalker. Then, we introduce the detailed designs including the walk matrix, benefit-aware I/O model, and block set-oriented walk updating scheme.

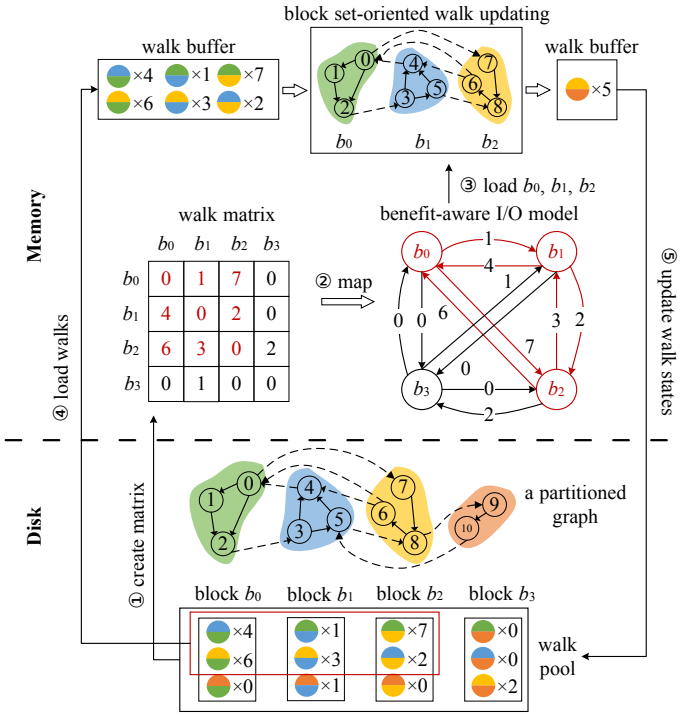


Figure 4: Overall design of SOWalker. View in color for optimal visualization.

3.1 System Overview

Figure 4 shows the overall design of SOWalker. To represent large graphs in the external memory setting, the graph is often partitioned into several blocks and stored on disks, and each block is associated with a walk pool storing the temporarily non-updatable walk states. A semi-circle in the walk pool is a vertex in a walk, with the color of the upper and lower semi-circle indicating the block that the previous and current vertices belong to, respectively. '$\times n$' means there are n walks of such state. Considering that the previous and current vertices may belong to two blocks, we propose a *walk matrix* to intuitively show the number of walks crossing between blocks. The values in the walk matrix are created and updated based on the walk states in the walk pool (①).

During the random walk procedure, we first load m blocks simultaneously to fit into memory, where m denotes the maximum number of blocks cached in memory. Suppose that $m = 3$ in Figure 4. We define loading a block as one *block I/O*, and scheduling and loading m blocks at the same time as a *batch*. To maximize accumulated updatable walks in a batch, we develop a *benefit-aware I/O model*. Specifically, relationships between blocks (i.e., the values in the walk matrix) can be mapped as a directed complete graph (②), and the block scheduling can be modeled into the maximum edge weight clique problem. Nodes in the clique are the blocks to be loaded. As an example in the figure, there are 23 updatable walks in block b_0, b_1, and b_2, which is the maximum

among all candidate block sets. Therefore, we load the three blocks into memory (③, shown as red nodes), and only load the walks whose previous and current vertices are both in the loaded blocks (④, walks in red box).

During the updating phase, we adopt a *block set-oriented walk updating scheme* that allows walks to access vertices in all loaded blocks since there are some edges connecting blocks (shown as arrows with dashed lines). That is to say, if the next vertex is still in memory, the walk can keep moving until it reaches a termination condition or visits a vertex belonging to a disk block. Here, 18 walks finish and the remaining 5 walks move to block b_3. When there are no more walks in memory, update the walk states in the walk pool (⑤). Repeat the process of block loading and walk updating until all random walks are finished.

3.2 Walk Matrix Representation

In order to skip loading non-updatable walks and eliminate useless walk I/Os, we use a walk matrix to represent the walks. The dimensionality of the matrix is the number of blocks. Each element (i, j) in this matrix represents the walks whose previous vertex belongs to block i, and the current vertex belongs to block j. The sum of all elements is the number of unfinished walks. The walk matrix is created and updated according to the walk states in the walk pool. In each batch, m blocks are selected based on the number of walks in the walk matrix, which will be discussed in Section 3.3. When all the walks in memory have been finished or have reached the boundary of the loaded block set, SOWalker checks the walk states to obtain the block IDs of the previous and current vertices. If the IDs are different, it means the walk is crossing blocks. Count the number of such walks that cross blocks and update the corresponding element of the walk matrix. Based on the walk matrix, SOWalker can readily check whether a walk can be updated, judging that both the previous vertex and the current vertex are in memory, thus skipping loading non-updatable walks and eliminating useless walk I/Os.

Figure 5 shows the detail of walk matrix W. The elements in W represent the number of walks crossing blocks at the current time. Suppose a graph is divided into 8 blocks, and the maximum number of blocks cached in memory is 3. If SOWalker selects blocks b_0, b_1, and b_2 to load into memory, then only the updatable walks, which are in the red box need to be loaded. Other walks do not need to be loaded because the blocks containing the previous or previous vertices are not in memory. Note that the number of walks in $W(i,i)$ is always 0 because the walk whose previous and current vertices are in the same block can be updated without additional block I/Os. Without the walk matrix, walks whose current vertices are in blocks b_0, b_1, and b_2 will be loaded (in the green box). However, the walks in the set difference of the green box and the red box are non-updatable walks.

In order to organize the walk data more compactly, we adopt a succinct data structure to encode each walk with 128 bits as shown in Figure 5 (on the right). *source*, *previous* and *current* is encoded in 29 bits, which represents the start vertex, previous vertex, and current vertex of a walk respectively. In this way, SOWalker can support starting random walks from 2^{29} source vertices simultaneously. Commonly, random walks are fed into downstream tasks, so it is necessary to save walk paths easily. In order to identify each walk quickly, we encode *walk ID* in 34 bits, which supports a maximum of 32 (i.e., $2^{34}/2^{29}$) walks starting from each vertex. Besides, *hop* indicates the number of steps the walk has already moved.

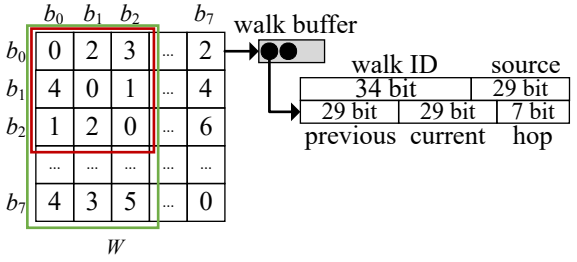


Figure 5: Walk matrix representation.

3.3 Benefit-Aware I/O Model

This section discusses how to efficiently schedule blocks. Since the previous and current vertices of a second-order random walk may belong to different blocks, we need to consider dependencies between blocks. Thus, we simultaneously schedule multiple blocks, instead of one block. Specifically, to improve the I/O utilization, we propose a benefit-aware I/O model to load multiple blocks with the maximum accumulated updatable walks. For this purpose, we formulate the block scheduling problem as the maximum edge weight clique problem. We also adopt an efficient heuristic algorithm to provide comparable I/O performance with a fraction of the cost.

Problem definition. Suppose the block set is B, $|B|$ is the number of blocks, and m is the maximum number of blocks cached in memory. $W(i, j)$ denotes the number of walks crossing between block i and j, that is, the values in the walk matrix. B_L is the loaded block set in a batch. The goal of block scheduling is to load multiple blocks with the maximum accumulated updatable walks, so the benefit is measured in terms of the *accumulated updatable walks* (*AUW*), defined as:

$$AUW(B_L) = \sum_{i \in B_L} \sum_{j \in B_L} W(i, j) \tag{1}$$

Problem mapping. Here, the relationship between blocks can be illustrated with a complete directed graph. The edge weight between two neighborhood blocks denotes the number of walks crossing between two blocks. For the block set B, the complete directed graph (*CDG*) is defined as:

$$CDG = (B, E) \tag{2}$$

$$E = \{e_{ij} = W(i,j) | i, j \in B\} \tag{3}$$

To maximize the accumulated updatable walks, we convert the block scheduling problem into the maximum edge weight clique problem, i.e., maximizing the sum of edge weights from all the feasible candidates.

Definition 1 (*Maximum Weighted Scheduling*, *MWS*): Given the complete directed graph $CDG = (B, E)$ along with the memory capacity requirements, MWS produces the loaded blocks, which satisfies that (1) the block scheduling problem is feasible according to the memory capacity requirements, i.e., the maximum number of blocks cached in memory; (2) the block scheduling problem produces the maximum sum of edge weights that maximizes the number of accumulated updatable walks in a batch.

Taking into account both the memory capacity and maximizing the sum of edge weights, the MWS is formulated as:

$$\max \sum_{i} \sum_{j} e_{ij} y_{ij} \tag{4}$$

$$\text{s.t.} \sum_{i=0}^{|B|-1} x_i = m \tag{5}$$

$$y_{ij} \leq x_i \tag{6}$$

$$y_{ij} \leq x_j \tag{7}$$

$$x_i, y_{ij} \in \{0, 1\} \tag{8}$$

where the variable x_i equals one if block i is loaded in memory. Constraint 5 guarantees that only m blocks can be chosen. By Constraints 6 and 7, for any edge (i, j), a binary variable $y_{ij} = 1$ if and only if both $x_i = 1$ and $x_j = 1$.

To maximizes $AUW(B_L)$, MWS must select m blocks for scheduling, but it ignores the contribution to block I/O reduction. In fact, the candidate block cached in memory does not yield block I/O but the walks in that block can be updated. Therefore, we aim to maximize the number of accumulated updatable walks in a block I/O. The objective function is redefined as:

$$\max_{B_L} S = \frac{AUW(B_L)}{k} \tag{9}$$

where k is the actual number of blocks to be loaded.

We use a bitmap β to record whether the block is cached in memory. If block i is in memory, the bit of block i is set to 1, i.e., $\beta_i = 1$. Otherwise, the bit is set to 0. By identifying which blocks are in memory, we can fully utilize the blocks in memory. Based on this consideration, we try to add the following constraint to the formulation:

$$\sum_{i=0}^{|B|-1} \beta_i \cdot x_i = m - k \tag{10}$$

where $m - k$ chosen blocks are already in memory. Constraint 10 ensures that the chosen block i does not need to be loaded, if and only if $\beta_i = 1$ and $x_i = 1$. We iterate over all $k \in [1, m]$ to find the optimal solution.

Algorithm 1: SA-based benefit-aware I/O model

Input: $CDG = (B, E)$, B_0: initial block set
Output: the loaded block set B_L

```
Function SelectBlocks(CDG=(B,E), B_0):
    B_L ← B_0                    // initial block set
    t ← T_0                      // initial temperature
    i ← 0                        // iteration counter
    while t ≥ T_s and i ≤ iter_max do
        B_c ← CHOOSENEWBLOCK(CDG, B_L)
        ΔS = S(B_c)-S(B_L)
        if ΔS>0 or e^(ΔS/t) > random(0,1) then
            B_L ← B_c
        t ← γt
        i ← i+1
    return B_L
```

The above linear programming method can guarantee the optimality of the solution obtained. However, the complexity of this problem is in order of 2^n [29]. As the scale of the problem is increasing, the complexity also soars. To settle the problem in a reasonable time, we adopt a heuristic algorithm to provide sub-optimal solutions, which is possible to solve large-scale problems within an acceptable time [30].

Solutions via heuristic algorithm. Since the maximum edge weight clique problem is NP-hard [31], many heuristic algorithms have been proposed to achieve a reasonable trade-off between computation time and solution quality. Simulated annealing (SA) is a local search procedure to find an efficient and feasible solution. In order to escape from local optima, a worse solution is accepted as the new solution with a probability that decreases as the computation proceeds. Despite its simpler structure and fewer parameters, SA has shown competitiveness in searching for optimal or near-optimal solutions and has been widely used to solve the maximum edge weight clique problem [32–35].

Inspired by Ernst et al. [35], we also use SA to select a loaded block set to maximize the number of accumulated updatable walks in a block I/O. The establishment of the objective function is described according to Equation 9. The detailed procedures of the SA-based benefit-aware I/O model is given as follows. Algorithm 1 illustrates the pseudo-code of this model.

Step 1: Initialize. Set initial temperature T_0, end temperature T_s, cooling coefficient γ of temperature, and the maximum number of iterations $iter_{max}$, where $iter_{max} = C^m_{|B|}$. Previous work has shown that a good initial solution results in faster convergence and improves the quality of the solution [36, 37]. In order to find a reasonably good initial block set, blocks appear in descending order of the number of walks in it. We

choose the top-m block as the initial block set B_0, meaning the block with more walks is more likely to be loaded into memory.

Step 2: Accept or reject the new solution. SA works iteratively by successively replacing the current solution with a random solution. In each iteration, randomly remove a selected block from the current block set B_L. Then, one of the remaining blocks is chosen randomly, and getting a new candidate block set B_c. Compute the difference between the new candidate block set B_c and the current block set B_L, i.e., the increment of the objective function $\Delta S = S(B_c) - S(B_L)$. If B_c is better, i.e., $\Delta S \geq 0$, then it will be accepted. Otherwise, it will be accepted with probability $p = e^{\Delta S/t}$, where t denotes the current temperature. Generate a random number ζ, where $\zeta \in [0,1]$. If $p > \zeta$, then B_c will be accepted. Otherwise, it will be rejected.

Step 3: Continue or end. Compute current temperature $t = \gamma t$ and the number of iterations $i = i + 1$. If $t < T_s$ or $i > iter_{max}$, end the algorithm. Otherwise, go back to step 2.

Compared to the exact but complicated linear programming method, SA provides an approximate solution but is much simpler, which yields orders of magnitude speedup and the computation time is only a small fraction of the total execution time (see Section 4.4.1).

3.4 Block Set-Oriented Walk Updating

Existing random walk systems partition a graph into several blocks. The block loading and walk management are at a block granularity, resulting in the walk updating being limited to a single block, called *block-oriented walk updating*. Once a walk reaches the boundary of the current block, the updating of a walk will be stopped. However, such a strategy hinders the walk updating and potentially increases the number of block I/Os.

To see this problem more concretely, we will consider a partitioned graph in Figure 4. The graph is partitioned into four blocks. Suppose two blocks are cached in memory and all walks start at vertex 0. We use the state-aware I/O model in GraphWalker [27] to load the block containing the largest number of walks as the current block, and iteratively load another block as the ancillary block. We skip loading the blocks without containing any previous vertex information. Figure 6(a) shows the process of block-oriented walk updating. (i, j) means the blocks cached in memory, where i is the current block, and j is the ancillary block. Only the walk in the current block can be updated. '+' means the lasted loaded block. As a result, 10 block I/Os are required and the walk steps per block I/O is 2.4.

We argue that although a graph is partitioned into several blocks, walks can move across blocks via the cut edges between these blocks. Thus, if a walk moves to any vertex belonging to the block in memory, it can further be updated, until it reaches the boundary of the block set in memory and moves to the block in disk. Such walk updating is called *block set-oriented walk updating*, which is illustrated in the example in Figure 6(b). In the first batch, blocks b_0 and b_1 are loaded into memory. Walk w_1 can be finished directly without extra block I/Os. While it needs 4 block I/Os in the block-oriented walk updating scheme. In the third batch, both walk w_0 and w_2 can be updated. In contrast, in the block-oriented walk updating scheme, only the walk in the current block can be updated. As a result, the number of block I/Os is reduced to 6 and the walk steps per block I/O increases to 4. The reason is that the block set-oriented walk updating allows walks to repeatedly visit the block in memory, which boosts the walk updating rate and accelerates the random walk process. Recently, GraSorw [28] also allows walks to be updated across the blocks. However, it limits the number of blocks in memory to 2, which is less flexible for different-scale graphs and random walks.

(i, j)	Walk paths
$(+b_0, +b_1)$	w_0: 0→7 w_1: 0→1→2→3 w_2: 0→2→3
(b_1, b_0)	w_1: 3→4→0 w_2: 3→5→8
$(+b_2, b_0)$	w_0: 7→8→6→4
$(b_2, +b_1)$	w_2: 8→6→0
$(+b_0, b_1)$	w_1: 0→2→3
$(b_0, +b_2)$	w_2: 0→1→2 (end)
$(+b_1, b_0)$	w_1: 3→4 (end)
$(b_1, +b_2)$	w_0: 4→0
$(+b_0, b_1)$	w_0: 0→7
$(+b_2, b_0)$	w_0: 7→8→9 (end)

(a) The process of block-oriented walk updating

(i, j)	Walk paths
$(+b_0, +b_1)$	w_0: 0→7 w_1: 0→1→2→3→4→ 0→2→3→4 (end) w_2: 0→2→3→5→8
$(b_0, +b_2)$	w_0: 7→8→6→4
$(+b_1, b_2)$	w_0: 4→0 w_2: 8→6→0
$(+b_0, b_1)$	w_0: 0→7
$(b_0, +b_2)$	w_0: 7→8→9 (end) w_2: 0→1→2 (end)

(b) The process of block set-oriented walk updating

Figure 6: Block- vs. Block set-oriented walk updating.

4 Evaluation

In this section, we evaluate the effectiveness of SOWalker. First, we introduce our experimental setup. Then, we compare SOWalker with two state-of-the-art random walk systems, GraphWalker [27] and GraSorw [28], in terms of overall performance and I/O efficiency. Third, we evaluate the effect of different block scheduling models. Finally, we analyze the impact of block size.

4.1 Setup

Environment. The hardware platform used in our experiments is a commodity server equipped with a 32-core 2.10 GHz Intel Xeon CPU E5-2620 with 128GB main memory and a 3TB HDD, running Ubuntu 20.04 LTS. SOWalker is implemented in around 4, 000 lines of C++ code and compiled by g++ 9.4.0 with an optimization flag as -O3. We use OpenMP for parallel random walks, and the number of threads is set to 32 unless explicitly specified. The reported results were averaged over 5 runs, and the error bars have been omitted as the variance was negligible.

Datasets. Table 2 describes the statistics of our evaluated graphs. RANDOM (RND) is a synthetic graph where each vertex is connected to five randomly selected neighbors. The probability of two vertices being connected is inversely proportional to the difference in their IDs. RMAT-27 (RM27), RMAT-28 (RM28), and Kron30 (K30) are synthetic graphs generated with the Graph500 generator [5], exhibiting a power-law degree distribution. Twitter (TW) [1] and Friendster (FR) [2] are social graphs that show the relationship between users within each online social network. UK-Union (UK) [3] and CrawlWeb (CW) [4] are web graphs that consist of hyperlink relationships between web pages. *Graph Size* is the amount of data stored in text format as an edge list. *CSR Size* is the storage cost to store graphs in CSR format. Since systems are executed in an out-of-core environment, the memory limit is set to 2GB (for RND and RM27), 4GB (for RM28, Twitter, Friendster, and UK-Union), or 32GB (for Kron30 and CrawlWeb). *Block Size* is heuristically set to 1/4 of the memory size according to Section 4.4.3. $|B|$ is the number of blocks that a graph is partitioned into according to the block size.

Dataset	$\|V\|$	$\|E\|$	Graph Size	CSR Size	Block Size	$\|B\|$
RM27	134.2M	1.1B	18GB	4GB	512MB	9
RND	268.4M	1.4B	24.7GB	5.2GB	512MB	11
TW	61.5M	1.5B	24.4GB	5.5GB	1GB	6
RM28	268.4M	2.1B	34.9GB	8GB	1GB	9
FR	65.6M	3.6B	58GB	13.5GB	1GB	14
UK	133.6M	5.5B	94.6GB	20.4GB	1GB	21
K30	1.1B	33.8B	628.3GB	120GB	8GB	16
CW	3.6B	126B	2.6TB	470GB	8GB	59

Table 2: Statistics of datasets.

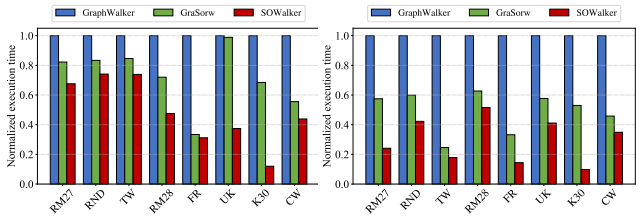
Graph algorithms. We evaluate SOWalker with two second-order random walk-based applications discussed in Section 2, i.e., node2vec and the second-order PageRank. For node2vec, we set the parameter $p = 0.5$, $q = 2$. Each vertex samples 10 walks with a fixed walk length of 80. For the second-order PageRank, the maximum walk length is 20, and we simulate 2, 000 random walks starting at each query source vertex.

Systems for comparison. We perform a comprehensive analysis of SOWalker's performance and compare it with two state-of-the-art random walk systems.

- GraphWalker [27], an I/O-efficient system for first-order random walks. When executing second-order random walks, we adopt the state-aware I/O model to load a block with the maximum number of walks as the current block and iteratively load another block into memory as the ancillary block. Both GraphWalker and SOWalker use the same parameter configuration.
- GraSorw [28] is the first out-of-core graph processing system designed for second-order random walks. It iteratively selects a block as the current block and uses a learning-based block loading model. However, this model consists of three stages: getting the running logs under the full-load mode, training, and running with the trained thresholds. Both the first and third stages involve second-order random walks, rendering it deficient in real-world applications. Therefore, we only use the full-load mode to load the ancillary block. On the other hand, GraSorw fixes the number of blocks in memory to 2, so we set the block size to half the memory size.

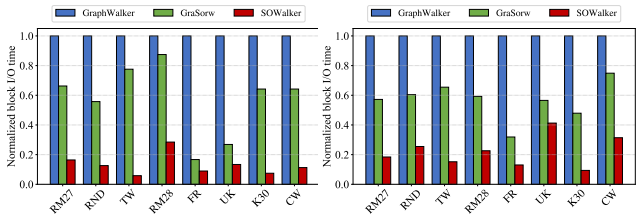
4.2 Overall Performance

We first compare the execution time of the chosen algorithms on different graphs and systems. Figure 7 shows the execution time normalized w.r.t. GraphWalker. We can see that SOWalker is faster than both GraphWalker and GraSorw in all cases. Specifically, SOWalker achieves 1.4-8.3× and 2.4-10.2× speedups over GraphWalker on node2vec and the second-order PageRank, respectively. As for GraSorw, SoWalker achieves 1.2-5.7× and 1.4-5.4× speedups over it on node2vec and the second-order PageRank, respectively. The main reason for the speedup in SOWalker is twofold. First, SOWalker loads multiple blocks with the maximum accumulated updatable walks, which improves the I/O utilization and the walk updating rate, so it requires much fewer blocks I/Os to run second-order random walks. GraSorw, on the other hand, is unaware of the walk states, just iteratively selects a block as the current block and loads an ancillary block into memory. Although GraphWalker loads a block with the maximum number of walks as the current block, it is unaware of the number of walks that can be updated. Therefore, both of them suffer poor performance. Second, SOWalker adopts the block set-oriented walk updating scheme, which allows



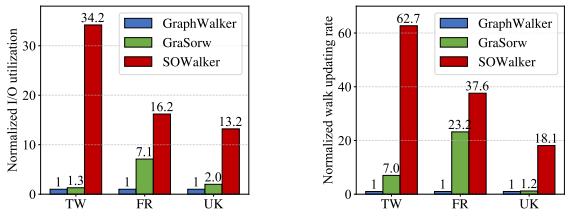
(a) Node2vec (b) Second-order PageRank

Figure 7: Execution time comparison.



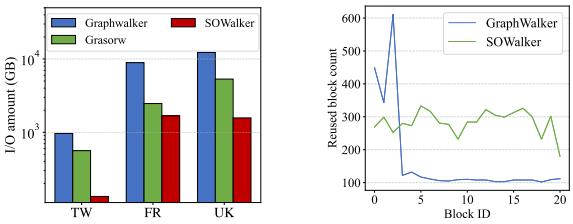
(a) Node2vec (b) Second-order PageRank

Figure 8: Block I/O comparison.



(a) I/O utilization (b) Walk updating rate

Figure 9: I/O utilization and walk updating rate.



(a) I/O amount (b) Reused block count

Figure 10: I/O amount and reused block count.

the loaded walks can be updated as much as possible in the loaded block set, so as to further accelerate the random walk process and reduce the block I/O costs.

4.3 I/O-efficiency Evaluation

Block I/O comparison. To justify the above argument, we compare the block I/O time on SOWalker and the other systems. Block I/O time is the time cost of loading blocks. Figure 8 shows the block I/O time normalized w.r.t. GraphWalker. In all cases, SOWalker outperforms both GraphWalker and GraSorw. Specifically, the block I/O time in SOWalker is only 5.8-41.3% of that in GraphWalker, and 7.5-72.9% of that in GraSorw, respectively. This is mainly attributed to SOWalker's benefit-aware I/O model that loads multiple blocks with the maximum accumulated updatable walks, so as to accelerate the random walk process and significantly reduce the block I/O number. On the other hand, GraphWalker and GraSorw load blocks iteratively, which incur great I/O cost.

I/O utilization. To verify that SOWalker can improve the I/O utilization, Figure 9(a) shows the average I/O utilization for node2vec on Twitter, Friendster, and UK-Union, normalized w.r.t. GraphWalker. As we can see, for all the graphs, SOWalker shows the highest average I/O utilization. Compared to GraphWalker and GraSorw, the I/O utilization of SOWalker is improved by 13.2-34.2× and 2.3-26.4×, respectively. This is mainly attributed to the benefit-aware I/O model, which maximizes the number of walks that can be updated, thereby improving I/O utilization. Besides, according to our walk matrix, we only loads walks whose previous and current vertices are both in memory. This guarantees that all walks can be updated and provides 100% walk utilization. While GraphWalker and GraSorw are unaware of the number of walks that can be updated, resulting in low I/O utilization.

Walk updating rate. Figure 9(b) reports the average walk updating rate normalized w.r.t. GraphWalker. The average walk updating rate in SOWalker significantly outperforms GraphWalker by up to 62.7×. Benefiting from our block set-oriented walk updating scheme, which allows each walk to move as many steps as possible in the loaded block set, SOWalker achieves more walk steps. In contrast, in GraphWalker, once a walk reaches the boundary of the block, the updating of a walk will stop, so the walk steps are limited. Although GraSorw also allows walks to be updated across the two blocks in memory, SOWalker still achieves up to 1.6-15.6× average walk updating rate stemming from the fact that it maximizes the I/O utilization based on the benefit-aware I/O model.

I/O amount. As mentioned earlier, the I/O amount can be divided into two parts: edge data on blocks and walk data. Let N be the total number of block I/Os, M be the block size, and W be the total number of loaded walks, with each walk encoded with 128 bits. The total I/O amount $A = N * M + W * 128$. Since block size is pre-defined, the I/O amount is proportional to the number of block I/Os and loaded walks. Figure 10(a) shows the I/O amount that each system runs node2vec on Twitter, Friendster, and UK-Union. We can see that there is a significant reduction in I/O amount. Compared to GraphWalker, SOWalker achieves an I/O reduction of over 80% on these graphs. Furthermore, the I/O amount in SOWalker

is only 23.7-67.9% of that in GraSorw. This reduction can be attributed to the accelerated random walk process, which enables the walks to be completed faster and significantly reduces the number of block I/Os.

Reused block counts. Blocks in memory can be reused as they do not yield block I/O but walks in these blocks can be updated. Figure 10(b) shows the reused counts for each block on UK-Union. The reused block counts in SOWalker are usually much higher than those in GraphWalker. The reason is that we consider the contribution of loaded blocks in the benefit-aware I/O model. In contrast, in GraphWalker, only a few blocks are highly reused, because many walks stay in these blocks. This makes them more likely to be selected as the current block and cached in memory for a long time. On the other hand, other ancillary blocks are iteratively loaded, resulting in frequent swapping between memory and disk. Consequently, the reused counts of these blocks are low.

4.4 Design Choices

In this section, we conduct experiments to validate some of our critical design choices that are essential to achieve optimal performance for SOWalker.

4.4.1 Comparisons of Scheduling Models

We now evaluate the effectiveness of the block scheduling models by comparing the following models:

- Random: randomly chooses m blocks to load into memory, which is used as the baseline.
- Max-m: chooses top-m blocks based on the number of walks in a block.
- Exact: the exact benefit-aware I/O model according to the linear programming method.
- Benefit-aware I/O model (BA): the benefit-aware I/O model according to the simulated annealing algorithm.

We run node2vec on UK-Union. A similar trend can also be observed on the other graphs and algorithms; their results are omitted due to space limitations. Table 3 presents the execution time, block I/O time, and computation time for the above models. Note that the computation time of the Random model and Max-m model is very short by a negligible amount. There are two observations that can be found. First, both the Random and Max-m models yield relatively higher execution times than BA model. This is expected since the Random model is an arbitrary order without any optimization. The Max-m model also suffers poor performance as it only focuses on the maximum number of walks. Some loaded walks cannot be updated due to the lack of previous vertex information, which wastes precious disk bandwidth and slows down the processing of random walks. Second, BA model achieves both the best performance and the near-optimal block scheduling model. To verify the correctness of BA, we compare it with the Exact model. The results exhibit that BA gives rise to a similar but slightly higher block I/O cost over the Exact model. More importantly, BA provides a speedup of 6.7× of the Exact model, and the computation time of the simulated annealing algorithm is only 10 seconds. While the computation time of the Exact model is nearly 3.5 hours, which constitutes 85% of the total execution time, and we cannot afford such a level of slowdown. In summary, our BA model can achieve faster runtime and better I/O performance.

Model	Execution time (s)	Block I/O time (s)	Block I/O number	Computation time (s)
Random	4970	3234	9868	-
Max-m	3871	2162	6391	-
Exact	14311	548	1484	12097
BA	2133	575	1537	10

Table 3: The comparison with different block scheduling models. '-' means that the computation time is negligible.

4.4.2 Comparisons of Walk Updating Schemes

Next, we evaluate the effectiveness of SOWalker's block set-oriented walk updating through a comparison experiment with block-oriented walk updating.

Block-oriented walk updating cannot be directly used in SOWalker, since our benefit-aware I/O model requires the help of block set-oriented walk updating. Therefore, we design a baseline system, which loads a block with the maximum number of walks as the current block and iteratively loads another block into memory as the ancillary block. We incrementally add the block set-oriented or block-oriented scheme to the baseline system and evaluate the performance impact of our contribution.

Figure 11 exhibits the performance of node2vec running on Twitter, Friendster, and UK-Union. The block set-oriented scheme outperforms the block-oriented scheme for all graphs.

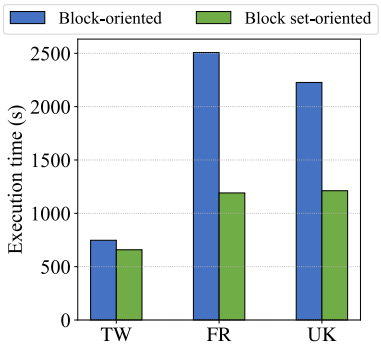


Figure 11: Block- vs. Block set-oriented walk updating.

The performance improvement is especially significant for Friendster, which yields up to 2.1× speedups. The reason behind this is that under the block set-oriented scheme, the walk can move across the entire loaded block set in memory. This leads to a longer walk steps in a block I/O. In contrast, the block-oriented scheme restricts walk updating to only one block, hindering the walk updating process and resulting in a large number of block I/Os.

4.4.3 Impact of Block Size

We also evaluate the impact on performance with different block sizes. Memory is limited to 4GB to illustrate the applicability. To demonstrate, we run node2vec on three representative graphs, Twitter, Friendster, and UK-Union. Figure 12 shows the execution time. The results on GraSorw are omitted since it fixes the number of blocks in memory to 2 and the block size is fixed to half the memory size. SOWalker presents superior performance over GraphWalker across all cases. This improvement is especially significant for small block sizes. This is because with the block-oriented walk updating scheme, GraphWalker restricts walk updating to a block, which severely wastes the vertex information in other blocks residing in memory. While our block set-oriented walk updating scheme allows walks to move across blocks in memory, so as to best utilize resources. Even when the block size is set to 2GB, i.e., 2 blocks in memory, the block-oriented walk updating scheme degrades into the block set-oriented walk updating scheme, the results are still encouraging. The reason is that our benefit-aware I/O model loads multiple blocks with the maximum accumulated updatable walks, so as to accelerate the random walk process. Besides, we observe that the block size should be neither too small nor too large to achieve a good performance in SOWalker. For the smaller block size, the walk updating rate is low. While for the larger block size, the I/O utilization is low. Therefore, according to our experiences, heuristically setting the block size to 1/4 of the memory size can produce the best performance.

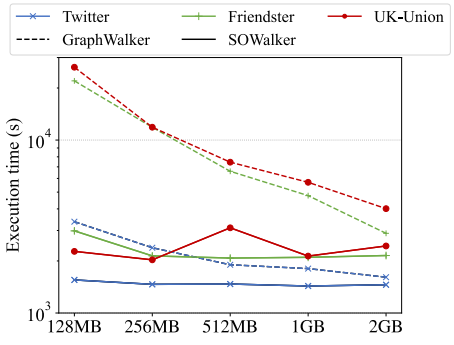
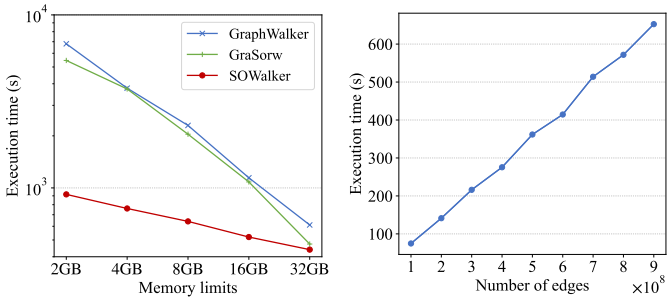


Figure 12: Impact of block size.



(a) Scalability with memory (b) Scalability with graph size

Figure 13: Scalability with memory and graph size.

4.5 Scalability

We evaluate the scalability of SOWalker by examining the performance improvements achieved with increased memory limits. Figure 13(a) illustrates the performance variance of node2vec on UK-Union as the memory increases. All three systems demonstrate good scalability when more powerful memory resources can be utilized. Although as memory increases, the performance disparities between systems tend to diminish, it is worth noting that SOWalker outperforms other systems utilizing 16GB memory, even when it is equipped with only 2GB memory. When the memory is increased to 32GB, the whole graph can fit into memory, and SOWalker still outperforms the other two systems, yielding 1.4× speedups.

We also evaluate the scalability of SOWalker with respect to graph size. We conduct experiments on random graphs and vary the number of edges from 1×10^8 to 9×10^8, with an average degree of 10. As shown in Figure 13(b), as the graph size increases, the implementation exhibits good scalability. However, due to the variability in the structures of different random graphs, some noise could be generated.

5 Related Work

Many graph systems have been proposed to process large graphs. In the past, numerous systems have emphasized the ability to run in a distributed environment, which use a cluster of machines to process large graphs [43–48]. However, distributed graph systems are still bugged by load imbalance problems and significant communication overheads.

Since out-of-core graph processing systems can represent large graphs in the external memory setting, they serve as a promising alternative to distributed solutions. GraphChi [22] is a pioneer in this category, which utilizes the Parallel Sliding Window (PSW) technique to reduce random I/O accesses from storage. X-Stream [49] provides a two-phase Scatter-Gather programming model that makes tradeoffs between random memory access and sequential access from streaming

data. GridGraph [23] presents a 2-level hierarchical partitioning scheme to improve the locality and reduce the number of I/Os. DynamicShards [24] uses dynamic shards to reduce disk I/Os. CLIP [25] and LUMOS [50] make full use of the loaded blocks to reduce disk I/O operations. However, these existing works were not originally designed for random walks and thus give sub-optimal performance.

With the increasing interest in the performance optimization of random walks, a large number of systems have been designed to handle random walks. DrunkardMob [26] is the first random walk system, which enables the simulation of billions of random walks on massive graphs, on just a single computer. However, it adopts the iteration-based model, which limits the efficiency and scalability of random walks. GraphWalker [27] develops a state-aware I/O model and an asynchronous random walk updating schedule to improve the I/O utilization. As it is designed for first-order random walks, it still incurs excessive disk I/Os when executing second-order random walks. GraSorw [28] is designed specifically for second-order random walks. It develops a bi-block execution engine and a learning-based block loading model to improve the I/O efficiency. However, its bi-block execution engine limits the number of blocks in memory to 2, which is less flexible for different-scale graphs and random walks. Moreover, the learning-based block loading model has to run the second-order random walk task twice to get the runtime statistics, rendering it deficient in real-world applications. SOWalker differs from all these systems in the walk representation, block scheduling model, and walk updating scheme. It designs a walk matrix to avoid loading non-updatable walks, proposes a benefit-aware I/O model to improve the I/O utilization, and adopts a block set-oriented walk updating scheme to boost the walk updating rate.

Meanwhile, memory optimizations and the increased number of cores make it possible to process large graphs more efficiently on a single machine. For example, ThunderRW [38] employs the step interleaving technique to hide memory access latency by switching the executions of different random walk queries. FlashMob [39] tries to harvest spatial and temporal locality underneath the apparently random nature of random walks. Besides, Shao et al. [40] proposed a memory-aware framework for second-order random walks, which automatically assigns a suitable sampling method for each node to minimize the time cost within a memory budget. While SOWalker focuses on I/O optimizations, some of these techniques can be implemented to further enhance the in-memory performance.

6 Conclusion

In this paper, we propose an I/O-optimized out-of-core graph processing system for second-order random walks, called SOWalker. To eliminate useless walk I/Os, we propose a walk matrix to prevent loading non-updatable walks. To improve the I/O utilization, we develop a benefit-aware I/O model to load multiple blocks with the maximum accumulated updatable walks. To boost the walk updating rate, we adopt a block set-oriented walk updating scheme to allow each walk to move as many steps as possible in the loaded block set. Our optimizations yield significant performance benefits compared to the state-of-the-art random walk systems and greatly reduce the I/O cost.

In the future, we would like to explore promising directions of second-order random walks. The current graph partitioning is quite simple, so we plan to design carefully graph partitions in order that random walkers should be trapped for long times in good partitions. Besides, we note that cache stall is also a performance bottleneck. The in-memory optimization of the second-order random walk is another attractive study to follow.

Acknowledgments

This work was supported in part by the Key Program of the National Natural Science Foundation of China (Grant No. 61832020), the Major Program of the National Natural Science Foundation of China (Grant No. 82090044), the Joint Funds of the National Natural Science Foundation of China (Grant No. U22A2027), and the Science Fund for Creative Research Groups of the National Natural Science Foundation of China (Grant No. 61821003). We are grateful to our shepherd, Călin Iorgulescu, and the anonymous reviewers for their constructive comments and suggestions.

References

[1] Twitter. http://an.kaist.ac.kr/traces/WWW2010.html.

[2] Friendster. https://snap.stanford.edu/data/com-Friendster.html.

[3] UK-Union. https://law.di.unimi.it/webdata/uk-union-2006-06-2007-05.

[4] CrawlWeb. http://webdatacommons.org.

[5] Graph500. https://graph500.org.

[6] Brad Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, Simone Rondelli, Alexander Ryazanov, Michael Schmidt, Kunal Sengupta, Bryan Thompson, Divij Vaidya, and Shawn Wang. Amazon Neptune: Graph Data Management in the Cloud. *ISWC Posters & Demonstrations, Industry and Blue Sky Ideas Tracks*, 2018.

[7] Spatial and Graph Analytics with Oracle Database 19c. Technical report, Oracle, 2019.

[8] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.

[9] Amazon Neptune. https://aws.amazon.com/cn/neptune.

[10] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.

[11] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast RandomWalk with Restart and Its Applications. In *Sixth international conference on data mining (ICDM)*, pages 613–622, 2006.

[12] Glen Jeh and Jennifer Widom. SimRank: A Measure of Structural-Context Similarity*. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 538–543, 2002.

[13] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 701–710, 2014.

[14] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 855–864, 2016.

[15] Amy N. Langville and Carl D. Meyer. Google's PageRank and Beyond: The Science of Search Engine Rankings. *Princeton university press*, 2011.

[16] Paul A.Gagniuc. Markov Chains: From Theory to Implementation and Experimentation. *John Wiley & Sons*, 2017.

[17] Denis R. Newman-Griffis and Eric Fosler-Lussier. Second-Order Word Embeddings from Nearest Neighbor Topological Features. *arXiv preprint arXiv: 1705.08488*, 2017.

[18] Wenyi Tang, Guangchun Luo, Yubao Wu, Ling Tian, Xu Zheng, and Zhipeng Cai. A Second-Order Diffusion Model for Influence Maximization in Social Networks. *IEEE Transactions on Computational Social Systems*, 6(4):702–714, 2019.

[19] Yubao Wu, Yuchen Bian, and Xiang Zhang. Remember Where You Came From: On The Second-Order Random Walk Based Proximity Measures. *Proceedings of the VLDB Endowment*, 10(1):13–24, 2016.

[20] Xueting Liao, Yubao Wu and Xiaojun Cao. Second-Order CoSimRank for Similarity Measures in Social Networks. In *2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[21] Martin Rosvall, Alcides V. Esquivel, Andrea Lancichinetti, Jevin D. West, and Renaud Lambiotte. Memory in Network Flows and its Effects on Spreading Dynamics and Community Detection. *Nature communications*, 5(1):1–13, 2014.

[22] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[23] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 376–386, 2015.

[24] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC)*, pages 507–522, 2016.

[25] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen and Weimin Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *2017 USENIX annual technical conference (USENIX ATC)*, pages 125–137, 2017.

[26] Aapo Kyrola. DrunkardMob: Billions of Random Walks on Just a PC. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 257–264, 2013.

[27] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 559–571, 2020.

[28] Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. An I/O-Efficient Disk-based Graph System for Scalable Second-Order RandomWalk of Large Graphs. *Proceedings of the VLDB Endowment*, 15(8):1619–1631, 2022.

[29] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[30] Ruizhi Li, Xiaoli Wu, Huan Liu, Jun Wu, and Minghao Yin. An Efficient Local Search for the Maximum Edge Weighted Clique Problem. *IEEE Access*, 6:10743–10753, 2018.

[31] Uriel Feige. Approximating Maximum Clique by Removing Subgraphs. *SIAM Journal on Discrete Mathematics*, 18(2):219–225, 2004.

[32] Sarab Almuhaideb, Najwa Altwaijry, Shahad AlMansour, Ashwaq AlMklaf, AlBandery Khalid AlMojel, Bushra AlQahtani, and Moshail AlHarran. Clique Finder: A Self-Adaptive Simulated Annealing Algorithm for the Maximum Clique Problem. *International Journal of Applied Metaheuristic Computing (IJAMC)*, 13(2):1–22, 2022.

[33] Xiutang Geng, Jin Xu, Jianhua Xiao and Linqiang Pan. A Simple Simulated Annealing Algorithm for the Maximum Clique Problem. *Information Sciences*, 177(22):5064–5071, 2007.

[34] Andrea Grosso, Marco Locatelli, and F Della Croce. Combining Swaps and Node Weights in an Adaptive Greedy Approach for the Maximum Clique Problem. *Journal of Heuristics*, 10(2):135–152, 2004.

[35] Ernst Althaus, Markus Blumenstock, Alexej Disterhoft, Andreas Hildebrandt, and Markus Krupp. Algorithms for the Maximum Weight Connected k-Induced Subgraph Problem. In *International Conference on Combinatorial Optimization and Applications*, pages 268–282, 2014.

[36] D. Janaki Ram, T. H. Sreenivas, and K. Ganapathy Subranmaniam. Parallel Simulated Annealing Algorithms. *Journal of parallel and distributed computing*, 37(2):207–212, 1996.

[37] Jun Gu and Xiaofei Huang. Efficient Local Search With Search Space Smoothing: A Case Study of the Traveling Salesman Problem (TSP). *IEEE Transactions on Systems, Man, and Cybernetics*, 24(5):728–735, 1994.

[38] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. ThunderRW: An In-Memory Graph Random Walk Engine. *Proceedings of the VLDB Endowment*, 14(11):1992–2005, 2021.

[39] Ke Yang, Xiaosong Ma, and Saravanan. Random Walks on Huge Graphs at Cache Efficiency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 311–326, 2021.

[40] Yingxia Shao, Shiyue Huang, Xupeng Miao, Bin Cui, and Lei Chen. Memory-Aware Framework for Efficient Second-Order RandomWalk on Large Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1797–1812m 2020.

[41] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[42] Lijun Chang. Efficient Maximum Clique Computation over Large Sparse. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 529–538, 2019.

[43] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[44] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Danny Bickson. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI)*, pages 17–30, 2012.

[45] 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 *11th USENIX symposium on operating systems design and implementation (OSDI)*, pages 599–613, 2014.

[46] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[47] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD)*, pages 135–146, 2010.

[48] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316, 2016.

[49] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, 2013.

[50] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 429–442, 2019.

Light-Dedup: A Light-weight Inline Deduplication Framework for Non-Volatile Memory File Systems

Jiansheng Qiu [†] [*] [§], *Yanqi Pan* [†] [*], *Wen Xia*[†]✉, *Xiaojia Huang*[†], *Wenjun Wu*[†], *Xiangyu Zou*[†], *Shiyi Li*[†], *Yu Hua*[‡]
[†]*Harbin Institute of Technology, Shenzhen* [‡]*Huazhong University of Science and Technology*
✉ *Corresponding Author: Wen Xia (xiawen@hit.edu.cn)*

Abstract

Emerging NVM is promising to become the next-generation storage media. However, its high cost hinders its development. Recent deduplication researches in NVM file systems demonstrate that NVM's cost can be reduced by eliminating redundant data blocks, but their design lacks complete insights into NVM's I/O mechanisms.

We propose Light-Dedup, a light-weight inline deduplication framework for NVM file systems that performs fast block-level deduplication while taking NVM's I/O mechanisms into consideration. Specifically, Light-Dedup proposes Light-Redundant-Block-Identifier (LRBI), which combines non-cryptographic hash with a speculative-prefetch-based byte-by-byte content-comparison approach. LRBI leverages the memory interface of NVM to enable asynchronous reads by speculatively prefetching in-NVM data blocks into the CPU/NVM buffers. Thus, NVM's read latency seen by content-comparison is markedly reduced due to buffer hits. Moreover, Light-Dedup adopts an in-NVM Light-Meta-Table (LMT) to store deduplication metadata and collaborate with LRBI. LMT is organized in the *region* granularity, which significantly reduces metadata I/O amplification and improves deduplication performance.

Experimental results suggest Light-Dedup achieves 1.01–8.98× I/O throughput over the state-of-the-art NVM deduplication file systems. Here, the speculative prefetch technique used in LRBI improves Light-Dedup by 0.3–118%. In addition, the region-based layout of LMT reduces metadata read/write amplification from 19.35×/9.86× to 6.10×/3.43× in our hand-crafted aging workload.

1 Introduction

Recently, Non-Volatile Memory (NVM) has been becoming increasingly popular. Its byte-addressability, persistence, and low latency enable it to be attached to the memory bus, sitting alongside the DRAM [24, 44, 52, 67]. Optane DC Persistent Memory Module (DCPMM) is the latest commercially available NVM. However, it is much more expensive than Hard Disk Drive (HDD) and Solid State Drive (SSD). Therefore, reducing the price of NVM is paramount for its future usage.

Deduplication, a system-level data compression approach, can enlarge the logical space and reduce the amortized cost of storage devices [16, 56, 62, 68]. Deduplication is widely used in file systems [72], backup systems [18–20, 39, 63], cloud computing [37, 57], etc. It usually calculates the fingerprints of data blocks and then identifies duplicates according to their fingerprints. For the redundant block, deduplication increments the reference count in the corresponding metadata to maintain data integrity.

Traditional disk-based deduplication approaches, such as using the cryptographic hash (e.g., SHA-256 and MD-5) to identify redundant data blocks, do not fit well with NVM since fast NVM has shifted the performance bottleneck from I/O to CPU. Prior works on deduplication for NVM [7, 28, 58, 75] propose several ways to address the issues. First, some works use offline deduplication to reduce the overhead of deduplication on the critical path, such as DeNOVA [28]. However, such background deduplication can neither enhance the file systems' write performance nor improve NVM's endurance. Second, many works use the non-cryptographic hash (e.g., CRC32 and xxHash [11]) to accelerate the identification of duplicate blocks. For example, NV-Dedup [58] leverages non-cryptographic hash to avoid calculating cryptographic hashes for most unique blocks, while DeWrite [75] shows that combing the non-cryptographic hash with byte-by-byte comparison is efficient for deduplication at cache line granularity.

Despite these efforts, existing works still fail to fully exploit the performance of NVM during deduplication due to a lack of comprehensive insights into NVM's I/O mechanisms. First, NVM's read/write asymmetry encourages researchers to combine non-cryptographic hash with byte-by-byte content-comparison to quickly identify the duplicate data [67, 75]. Thus the overheads of cryptographic hash calculation can be eliminated. Second, we observe other two NVM I/O features that hinder NVM deduplication performance: (1) *Long media read latency*. Despite its read/write asymmetry, NVM's read latency is 2–3× higher than the write since the write buffer inside NVM hides the long media write latency [67]. Therefore, there is still a large room left for the acceleration of content-comparison by hiding the read latency. (2) *Coarse media access granularity*. The mismatch between the size of deduplication metadata (commonly 16–64 bytes for each data block) and the coarse media access granularity in NVM (e.g., 256 bytes XPLine of DCPMM) can lead to severe metadata I/O amplification if the access to the metadata lacks locality,

[*] Jiansheng Qiu and Yanqi Pan are co-first authors of the paper.
[§] Now working at Tsinghua University.

which degrades not only deduplication performance but also NVM's endurance, especially when the system is aged.

This paper presents Light-Dedup, a novel light-weight inline deduplication framework for NVM file systems. Light-Dedup is designed with two specific goals in mind: (1) Maximizing the deduplication performance by considering NVM's memory interface, read/write asymmetry, and access granularity, while adding negligible overhead to the critical path. (2) Retaining low deduplication metadata I/O amplification even if the file system is severely aged (i.e., many holes).

To achieve the first goal, Light-Dedup proposes Light-Redundant-Block-Identifier (LRBI) to quickly identify the duplicate blocks. Unlike the prior works that use both non-cryptographic and cryptographic hash [58] or that straightforwardly combine non-cryptographic hash with byte-by-byte content-comparison [75], LRBI considers both NVM's read/write asymmetry and long media read latency in redundant block identification. Specifically, LRBI uses xxHash, one of the fastest non-cryptographic hashes [11], to quickly identify most non-duplicate blocks. For those blocks with the same fingerprint, LRBI leverages NVM's memory interface to enable asynchronous NVM reads and proposes speculative prefetch to minimize the read latency seen by content-comparison. In particular, speculative prefetch uses *In-Block* and *Cross-Block Prefetch* to exploit the parallelism between NVM read and CPU computation.

To achieve the second goal, Light-Dedup organizes its in-NVM deduplication metadata table, Light-Meta-Table (LMT), as a *region*-based linked list. Each *region* contains multiple continuous metadata entries. Each entry stores the critical information for both basic deduplication and speculative prefetch used in LRBI. The allocation of metadata entries is done first by allocating a *region* and then by allocating entries in that region almost sequentially, which significantly reduces the deduplication metadata I/O amplification caused by NVM's coarse access granularity, especially in an aged file system. In addition, LMT trades 1× extra deduplication metadata space usage for zero garbage collection overheads, which retains the stabilization of deduplication performance.

In summary, this paper makes the following contributions:

- We perform an in-depth analysis of how deduplication can be affected by several NVM's I/O mechanisms and introduce how to maximize NVM deduplication performance with full consideration of them.
- We propose an inline deduplication framework for NVM file systems, Light-Dedup, with two key techniques: (1) LRBI combines non-cryptographic hash with speculative-prefetch-based content-comparison to fully leverage NVM's I/O asymmetry while hiding its media read latency by enabling asynchronous NVM reads. (2) The *region*-based layout is adopted in LMT to manage deduplication metadata with a good locality and retain low metadata I/O amplification.
- We implement Light-Dedup in Linux kernel 5.1.0 based on NOVA [66], one state-of-the-art NVM file system. The code is available at https://github.com/Light-Dedup/Light-Dedup. Furthermore, we make a comprehensive evaluation of various synthetic and real-world workloads. The results show that Light-Dedup adds negligible overhead while significantly improving the file system's write performance under a high duplication ratio.

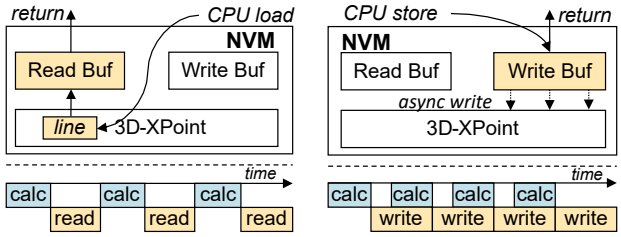
2 Background and Related Work

2.1 NVM and NVM File Systems

With its byte-addressability, low latency, persistence, and low power consumption [3, 21, 26, 35, 40, 41, 46], NVM becomes a promising candidate for next-generation storage media. In this work, we focus on high-density storage-type NVM with the memory-like interface [29] that serves as persistent storage media. For brevity, we denote such storage-type NVM as NVM. According to the latest research on the commercial DCPMM [64, 67] and our investigation on existing NVM devices [6, 34, 50, 51, 65, 71], this paper concludes the following **five common I/O features** of NVM that potentially have impacts on NVM deduplication performance:

- *Asymmetry in Read/Write Bandwith*. The read bandwidth of NVM is up to 3× than its write [67]. The feature is common for persistent storage media such as Phase Change Memory (PCM) [50, 71], STT-RAM [6, 34], memristor [65], 3D-XPoint [67], NAND flash [1], etc.
- *I/O with Buffers*. For writes, NVM leverages the buffer to write asynchronously to hide long media write latency. While for uncached reads, NVM fetches data synchronously from the media. The data will be cached in the internal read buffer for future reads [51, 64, 67]. Figure 1 shows the I/O mechanisms of NVM.
- *Coarse Access Granularity*. Coarse media access granularity is common for storage-type NVM. For example, the row buffer size of a PCM is preferred to be larger than 128 bytes [30]. Coarse access granularity (and the above I/O buffers) is beneficial for improving storage bandwidth and bridging the performance gap between storage and CPU. Since NVM is denser but slower compared to DRAM, it is reasonable that NVM has a larger access granularity than a cache line.
- *Long Media Read Latency*. The underlying non-volatile media generally introduces relatively longer media latency than DRAM [29, 30, 43, 54]. The synchronous data fetch mechanism fails to hide such latency [64].
- *Memory Interface*. With the memory interface, NVM can be accessed by CPU store/load. This feature makes asynchronous CPU prefetch possible, which can be leveraged to address NVM's long media read latency.

To well exploit the physical characteristics of NVM devices, several NVM file systems [8, 12, 17, 70] are proposed. Among these file systems, NOVA [66] is the state-of-the-art one, which aims to exploit the potential of DRAM and NVM



(a) Synchronous reads (b) Asynchronous writes

Figure 1: NVM I/O buffering mechanisms.

hybrid memory systems while providing strong consistency guarantees. Specifically, NOVA allocates a separate log for each inode (i.e., file), appends single inode operations as entries of the inode's log, and atomically updates the log's tail to commit the operations. For operations involving multiple inodes, NOVA records the log tail's pointers of the affected inodes in the journal to update them atomically. Further, NOVA accelerates search operations by maintaining radix trees of directories and files in DRAM.

2.2 Inline Deduplication Techniques for NVM

File system deduplication [62] is a block-level redundancy elimination technique and has been needed in many applications [18–20, 59, 63, 72]. This paper focuses on inline deduplication since the offline approach neither enhances file systems' write performance nor improves NVM's endurance. Typically, an inline deduplication framework consists of several techniques, including redundancy identification, deduplication metadata management, indexing, etc.

Redundant Block Identification Techniques. Traditional disk-based deduplication approaches, such as using the cryptographic hash to determine the duplicates [16, 56, 68], do not fit well with fast NVM devices due to their heavy software overhead. Recent works [7, 58, 75] leverage non-cryptographic hash functions to reduce the computation cost of fingerprints. However, non-cryptographic hash suffers from hash collision (different blocks have the same hash). To address the issue, existing approaches either apply cryptographic hash (e.g., NV-Dedup [58]) or straightforwardly perform byte-by-byte content-comparison (e.g., DeWrite [75]) to verify if the blocks are duplicate when their non-cryptographic hashes equal[1]. However, (1) cryptographic hash calculation is a bottleneck when there are many duplicates; (2) the long latency of uncached reads hinders the performance of content-comparison, especially when the concurrency level is low and read/write asymmetry is not obvious. In contrast, our approach (i.e., LRBI) combines non-cryptographic hash with a speculative-prefetch-based content-comparison technique to exploit I/O asymmetry and hide long media read latency.

In-Storage Deduplication Metadata Management. To manage redundant blocks, deduplication approaches must maintain in-storage structures to store the basic information about the deduplicated blocks (e.g., the mappings between fingerprints and physical blocks, reference count, etc.). Besides, some additional bits are required for the collaboration with redundancy identification. Existing NVM deduplication approaches (e.g., NV-Dedup [58], LO-Dedup [7], DeWrite [75]) often reserve a fixed in-NVM table to store the deduplication metadata and to allocate/free them by the free list. Additionally, NV-Dedup maintains both non-cryptographic and cryptographic hash in the table; LO-Dedup organizes the deduplication metadata as an ordered linked list structure to accelerate the continuous matching. However, such free-list-based management is not NVM-friendly since its allocation strategy can introduce significant fragmentation when the system is aged, causing a severe metadata I/O amplification. In contrast, our proposed LMT manages the deduplication metadata as a *region*-based linked list, provides course-grained metadata management, and significantly reduces metadata I/O amplification under aged file systems.

In-NVM Deduplication Metadata Index Techniques. Searching for in-NVM deduplication metadata based on the calculated fingerprint is a critical step in deduplication. To prevent frequent NVM accesses, existing works usually build an in-DRAM index, such as static hash tables or red-black tree, to accelerate the search [7, 58]. We use a dynamic hash table (i.e., *rhashtable* [13]) for its resizability and efficiency.

3 Observations and Motivations

3.1 Data Redundancy & NVM Deduplication

Data redundancy is a common phenomenon in storage systems with the exponential growth of data. Prior works [19, 20, 63, 72] have observed a large number of redundancies in modern primary storage systems. For example, there are 95% and 47% duplicates (in 4 KiB block granularity) in two real-world traces collected by FIU: *Mails* and *WebVMs* [27, 33]. Thus, the deduplication approach is a promising solution to enlarge logical storage space and reduce storage costs. As the next-generation storage media, deduplication for expensive NVM is profitable and urgent. Recently, many research efforts have designed deduplication schemes specifically for NVM file systems [7, 58]. However, they fail to fully exploit the characteristics of NVM's I/O mechanisms and leave substantial room for improvement from the performance point of view. This paper aims to develop a more efficient inline deduplication framework scheme for NVM file systems that can fully exploit the I/O characteristics of NVM devices.

3.2 I/O Asymmetry and Read Latency in NVM Redundant Block Identification

The efficiency of redundant block identification is essential to NVM deduplication performance. Traditional disk-based deduplication approaches use the cryptographic hash to identify the duplicate blocks [16, 32, 56, 68]. However, it does not suit NVM deduplication well due to its computation overhead, which wastes much CPU computation and thus starves

[1] LO-Dedup does not address the hash collision of the non-cryptographic hash used in their paper. Thus, we omit its redundancy identification technique.

Table 1: The breakdown deduplication time. Light denotes Light-Dedup, and *LD-w/o-P* denotes a simple deduplication file system that incorporates non-cryptographic hash with content-comparison into the write path and deduplicates 4 KiB blocks. With the introduced speculative prefetch technique (i.e., Light), content-comparison time is dropped by 62.2%.

System	Calc. Lat (ns)		I/O Lat (ns)		Bandwidth (MiB/s)
	fp	others	write	cmp	
NOVA	0.0	84.7	2275.6	0.0	1401
LD-w/o-P (1st)	309.9	1072.5	**585.3**	0.0	1612
LD-w/o-P (2nd)	308.0	571.6	0.0	**3263.0**	870
Light (1st)	310.0	1131.3	**559.8**	0.0	1592
Light (2nd)	0.0	343.3	0.0	**1234.8**	1914

NVM. We observe the problem in NV-Dedup [58]. In a simple sequential write 4 GiB workload, the write bandwidth of NV-Dedup drops by 52.5% with the duplication ratio increasing from 0% to 75%. The root cause is that cryptographic hash calculation (i.e., MD-5) dominates up to 64.9% of the whole write time since NV-Dedup relies on the cryptographic hash to handle its non-cryptographic hash collision.

To address the above heavy computation, DeWrite [75] uses non-cryptographic hash and byte-by-byte comparison in the combined manner [10]. The method is well aligned with the characteristic of NVM since it prevents heavy CPU computation and leverages the large read/write asymmetry to trade slow duplicate writes for faster reads (i.e., content-comparison). However, recent researches about NVM reads [64, 67] show that they can still be a bottleneck since uncached reads have to fetch data from media synchronously, which introduces long media read latency and thus negatively affects the content-comparison performance.

We examine how exactly NVM I/O affects the deduplication performance. Table 1 shows the breakdown latency of orderly writing two 4 GiB files with identical content (2 MiB per I/O) to NOVA and *LD-w/o-P*[2] under a single thread. Note that the first write conducts no duplicate blocks, but the second write causes 100% duplicates and results in reads (caused by content-comparison). We observe several interesting phenomena from Table 1. First, *LD-w/o-P* has a surprisingly low write time (585.3ms) compared to NOVA (2275.6ms) because asynchronous write enables the parallelism between CPU computation and write I/O. Thus, computation hides part of NVM writes latency (as shown in Figure 1b). Second, during the second writes, the write bandwidth of *LD-w/o-P* drops by 46% compared to the first. We find that the content-comparison time arises to 3263.0ns, which dominates 78.8% deduplication latency. The above observations suggest that non-cryptographic hash-based redundant block identification adds negligible overheads to the normal non-deduplication write path (i.e., the data blocks to be written are all unique). However, deduplication performance is significantly limited by the long read latency and is far from ideal. We believe there are two reasons: (1) NVM's read/write asymmetry under low thread count is not large enough [67]. (2) Current hardware prefetcher of intel 64 bits architecture fails to remedy the drawbacks of NVM's long media read latency since it is designed for DRAM and only attempts to prefetch two cache lines ahead of the prefetch stream [23].

[2]We build *LD-w/o-P* based on our proposed Light-Dedup by removing the speculative prefetch technique. This means that the deduplication metadata management and indexing are the same as in Light-Dedup, but they have negligible performance impacts on the experiments of this subsection.

Table 2: The average NVM extra reads/writes of deduplication metadata for writing each block.

Approaches	First Write		Second Write	
	Read (B)	Write (B)	Read (B)	Write (B)
ideal	≈40	40	40	≈40
All-in-NVM	726.12	293.17	528.65	259.05
Entry-based	126.94	79.56	774.13	394.54
Ours	116.28	75.75	244.19	137.17

In summary, NVM's long read latency hinders content-comparison performance during NVM deduplication. Considering NVM's asynchronous writes and the limitations of hardware prefetcher, we are motivated to think: Can we manually achieve asynchronous reads to hide media read latency? Memory characteristics of NVM inspire us to obtain our first motivation: We can leverage memory prefetch instructions to enable asynchronous NVM reads and thus accelerate content-comparsion. However, applying prefetch to NVM deduplication is not straightforward. There are two technical challenges: (1) The limited number of concurrent prefetch instructions that a CPU core can handle. (2) How to incorporate the prefetch mechanism into deduplication logic.

3.3 Metadata I/O Amplification in NVM Deduplication Metadata Management

During NVM deduplication, deduplication metadata can be frequently accessed and updated, causing a large amount of small NVM accesses. Metadata I/O amplification will get larger if these small NVM accesses exhibit a random pattern due to NVM's coarse access granularity. However, existing NVM deduplication file systems pay little attention to the issue. In this subsection, we investigate two widely used NVM deduplication metadata management approaches.

All-in-NVM Management. DeNOVA [28] takes an All-in-NVM design and constructs an in-NVM hash table to store and index the deduplication metadata to reduce DRAM consumption. However, hash tables typically exhibit random access patterns [36, 45, 64, 73, 74], which leads to severe read/write amplification. To verify this, we implement an inline deduplication system with an All-in-NVM design based on *Light-Dedup*: Its deduplication metadata are organized as a static hash table in NVM. We write the same 64 GiB file twice and measure the extra NVM reads/writes of deduplication metadata using *ipmctl* [22]. The experimental results are shown in Table 2. Ideally, each block write results in about 40 bytes NVM read/write (i.e., 32 bytes for deduplication

metadata entry, 8 bytes for the mapping from block number to metadata entry, see §4.2). In this case, read and write amplification are $528.65/40 \approx 13.2\times$ and $293.17/40 \approx 7.3\times$. Note that the reason why read amplification is nearly $2\times$ of theoretical upper bound ($8\times$) may be due to the prefetch mechanism and buffering strategy of internal Optane DCPMM hardware [64].

NVM-DRAM Hybrid Entry-Based Management. NV-Dedup [58] and LO-Dedup [7] store the deduplication metadata in NVM and maintain the in-DRAM index to locate them efficiently, which alleviates the problem of all-in-NVM design. Their deduplication metadata is managed at the granularity of cache lines, aligned in a manner favored by CPUs, and allocated/freed through a free list. We refer to this approach as *entry-based*. It is acceptable for a fresh new file system, but when the file system is aged, the physical location of allocated free entries can be random, which results in random access to NVM and causes severe read/write amplification. To show the problem, we make an extensive evaluation in §5.5 and present part of the results in the *Entry-based* row in Table 2, in which the first write is performed in the fresh new system and the second write is in the aged system. The results suggest that such entry-based metadata management can lead to significant read/write amplification in the aged system, about $774.13/40 \approx 19.35\times$ and $394.54/40 \approx 9.86\times$, respectively.

In summary, the severe metadata I/O amplification observed in Table 2 wears out NVMs and leads to performance degradation under aging file systems. To alleviate the problem, we focus on redesigning the hybrid deduplication metadata management strategy. Inspired by mimalloc [31], which shards its free list in page granularity, we obtain our second motivation: managing deduplication metadata in the *region* (i.e., 4 KiB block) granularity to maintain access locality, which elegantly reduces metadata I/O amplification. However, the issue of how to reclaim stale entries (i.e., garbage collection) with minimal overhead and design entry fields that collaborate with LRBI remains unresolved.

4 Design and Implementation

4.1 System Overview

Based on the observations of NVM's internal I/O mechanisms, we propose Light-Dedup, a light-weight inline deduplication framework for NVM file systems, as shown in Figure 2. It includes two key techniques:

- **Light-Redundant-Block-Identifier (LRBI)**. LRBI is proposed to quickly identify duplicate blocks by exploiting NVM's large read/write asymmetry and hiding long media read latency. It combines non-cryptographic hash with a speculative-prefetch-based byte-by-byte content-comparison technique. Specifically, speculative prefetch leverages NVM's memory interface and uses *In-Block* and *Cross-Block Prefetch* techniques to asynchronously load speculated data into CPU/NVM buffers, which exploits the parallelism of NVM I/O and CPU computation and thus markedly hides read latency.

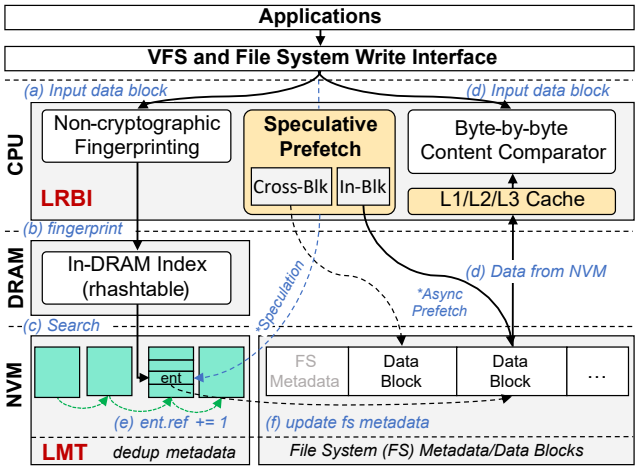


Figure 2: Light-Dedup overview.

- **Light-Meta-Table (LMT)**. LMT is an in-NVM table responsible for (1) storing basic deduplication metadata, such as the mapping from fingerprints to physical blocks; (2) maintaining speculation information, such as the hint of where and whether to prefetch the to-be-compared block. LMT adopts *region-based layout* to retain the locality of deduplication metadata. A *region* is a 4 KiB block, and *regions* are linked by 8 bytes pointers. Each dedup metadata is allocated in the *region* almost sequentially, hence reducing metadata I/O amplification. In addition, to prevent GC overheads brought by sequentiality, LMT trades $1\times$ more space for zero GC overheads.

In-DRAM Index. Light-Dedup adapts a *rhashtable* in DRAM to locate in-NVM deduplication metadata entry, whose key is the hash value (i.e., fingerprint) and value is the in-NVM position of the corresponding entry. We use *rhashtable* since it is a mature, well-tested, and efficient dynamic hash table implementation in the mainline Linux kernel [13], which suits for point-query (i.e., searching for a specific deduplication metadata entry by the given fingerprint). Meanwhile, *rhashtable* leverages efficient Read-Copy-Update (RCU) Lock [38] to handle the concurrent accesses to LMT. Note that this paper does not aim to redesign a specific index structure since in-DRAM hash table indexing is commonly efficient (e.g., usually achieves constant access time), and there have been many works dedicated on this [36, 45, 73, 74].

Put NVM I/O Features & Light-Dedup Together. We summarize Light-Dedup's insights into the presented NVM I/O features. First, read/write asymmetry can be leveraged to improve NVM deduplication performance by combing non-cryptographic hash with content-comparison, trading the slow duplicate writes for the faster reads (§4.2). Second, this paper observes that long media read latency hinders content-comparison performance dramatically. To address the problem, we propose LRBI, which follows the non-cryptographic hash-based infrastructure but leverages NVM's memory inter-

face and I/O buffering to enable asynchronous reads, and thus markedly accelerates content-comparison (§4.3). Third, Light-Dedup addresses the metadata I/O amplification brought by NVM's coarse access granularity by organizing LMT as a *region*-based linked list (§4.4).

4.2 Basic Deduplication Logic

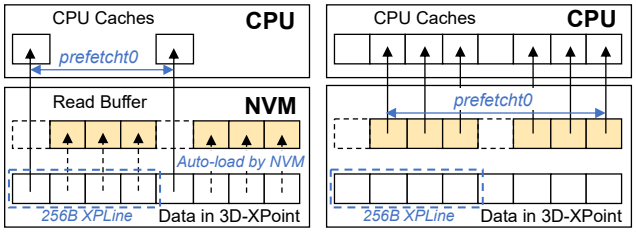
This subsection will introduce the basic deduplication flows of Light-Dedup (without speculative prefetch) to show how non-cryptographic hash and byte-by-byte comparison can be integrated into NVM file systems. Notably, we refer to this version as *LD-w/o-P*, as mentioned in §3.2.

Write Logic. Assume that the file system is writing a 4 KiB duplicate block. As shown in Figure 2, during the write, (1) Light-Dedup first calculates the non-cryptographic hash as the fingerprint of the input block (see steps (a), (b)), (2) and then efficiently locates the corresponding deduplication metadata entry (refer to as entry for brevity) in LMT by searching for the given fingerprint in *rhashtable* (see step (c)). (3) Once the entry is found (i.e., hash value matches), Light-Dedup compares the content of the input block and the block corresponding to the found entry byte-by-byte (see step (d)). (4) Finally, assuming comparison determines that the input block is duplicate, Light-Dedup increments the reference count of the duplicate block and records its duplicate block number in the file system metadata (see steps (e), (f)).

There are two exceptional cases to be handled. Given the fingerprint of the input block, (1) if no entry with the same fingerprint is found in LMT, suggesting that the input block is unique, then it will be written normally by the file system. After that, both its fingerprint and the block number are stored in a newly allocated entry with the reference count set to one. (2) If an entry is found, but the content of the stored block and the input one are different, then the input block becomes a non-dedup block, i.e., Light-Dedup does not allocate an entry for it, which will not affect the correctness of the deduplication system since only the file system has access to that block.

Deletion Logic. Light-Dedup maintains another in-NVM table that maps the block number to the offset of the corresponding deduplication metadata entry (i.e., similar to an inverse index). Note that the mapping maintenance introduces another 8 Byte metadata write in the write path (this is why the ideal NVM access is 40 bytes instead of 32 bytes). To delete a block, Light-Dedup first locates its deduplication metadata entry with the aforementioned table. If the entry is not found in the table, suggesting that the block number has not been inserted into the metadata table, then Light-Dedup frees the block directly. Otherwise, Light-Dedup decreases the reference count of the block (recorded in the entry) by one. If the reference count becomes zero, then the block can be safely freed. Deleting blocks in Light-Dedup does not cause the garbage collection of deduplication metadata due to the tradeoff in LMT (see §4.4).

Read Logic. The read path remains the same as the non-dedup file systems. Note that deduplication fragments files' data, which may lead to random reads. However, Light-Dedup deduplicates 4 KiB blocks and large random access to NVM does not cause significant performance degradation [64].



(a) 1st: Prefetching XPLines. (b) 2nd: Prefetching read buffer.

Figure 3: *In-Block Prefetch (IBP)* mechanisms. Each square in the figure indicates one 64 bytes cache line.

4.3 LRBI: Dedup with Speculative Prefetch

In this subsection, we present the step-by-step design of speculative prefetch used in LRBI, which aims to reduce the read latency seen by content-comparison (§3.2). Its key design principle is to enable asynchronous NVM reads and to markedly improve the parallelism between NVM I/O and CPU computation. It consists of two prefetch strategies: *In-Block Prefetch* and *Cross-Block Prefetch*.

4.3.1 In-Block Prefetch (IBP)

IBP speculates that the content-comparison always tends to compare all bytes and leverages memory prefetch instructions, e.g., `prefetcht0` assembly instruction in x86 [23], to enhance the parallelism of reading bytes in the same block.

Prefetch-Cmp-64 (P64). The most straightforward prefetch strategy is issuing 64 prefetch instructions to load 64 cache lines of the block into CPU caches (i.e., $64 \times 64 = 4096$ bytes) before comparing it with the input block (if they have the same fingerprint value). However, the maximum number of concurrent prefetch instructions a CPU core can handle is limited. In our machine, that number is in the open range $(8, 16)$. Therefore, many prefetch instructions in *P64* are executed in a serial manner, and thus the parallelism is limited.

In-Block Prefetch (IBP). We leverage the large access granularity of NVM (e.g., 256 bytes XPLine) to address the problem of *P64*. As Figure 3 shows, first, we issue 16 prefetch instructions with stride 256 bytes to touch the first cache line of XPLine, and the whole block is loaded automatically into CPU caches or NVM read buffer [64], where most of the NVM read is in parallel. Second, we issue prefetch instructions with stride 64 bytes to bring the in-read-buffer data into CPU caches. Note that *In-Block Prefetch* is a general optimization technique that can be applied to some other block-based NVM I/O scenarios, but it is beyond the scope of this paper.

4.3.2 Cross-Block Prefetch (CBP)

Unlike *IBP*, *Cross-Block Prefetch* exploits the parallelism among CPU tasks (i.e., fingerprint calculations and content-comparison, etc.) and NVM I/O tasks (i.e., reading a to-be-compared data block), and thus hides NVM's media read

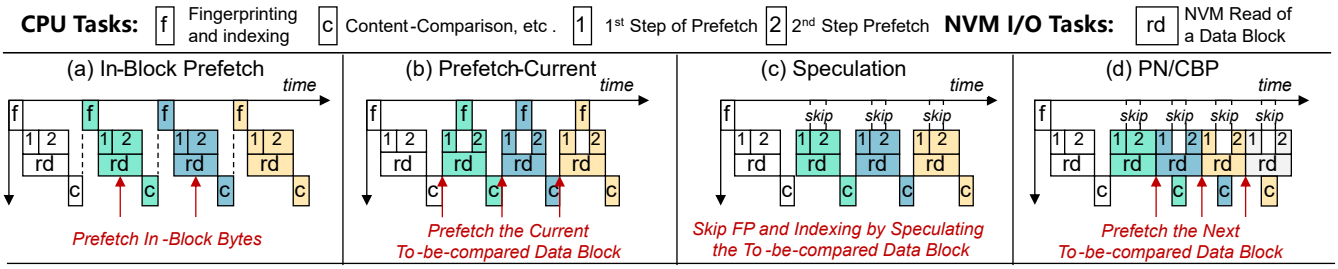


Figure 4: Simplified time sequence of Light-Dedup using different prefetch mechanisms: *IBP* improves parallelism of reading an NVM block while *PC*/*SP*/*PN*/*CBP* further improves the parallelism between CPU and NVM I/O tasks. Here, task c includes content-comparison, updating dedup/FS metadata, etc. Note that the time of running task 2 depends on the amount of data loaded into NVM's read buffer. Thus, task 2 can be faster (shorter) after running tasks f and c in *PC* and *PN*/*CBP*, respectively.

latency. This subsection introduces the technique with step-by-step explorations, as shown in Figures 4(b)–(d).

Determining Where and Whether to Prefetch the To-be-compared Data Block. This is critical for Light-Dedup to support *Cross-Block Prefetch*. First, we incorporate a "hint" field into the deduplication metadata entry. The hint maintains both the location of the subsequent speculated block's entry and the *trust degree* (range: $[0,7]$) that indicates if the speculation is correct. Second, to maintain the hint, we keep track of the entry of the last written block and check the corresponding hint after the deduplication of the current input block finishes: *trust degree* is decreased by 2 if the hint is proven to be wrong, and increased by 1 if correct. The hint is trusted and followed only if the trust degree reaches the trust threshold (i.e., 4 for now). Third, since prefetch consumes NVM read bandwidth, we are conservative about its usage. Thus, we further introduce *per-CPU stream trust degree* to indicate the locality of the workload, which is increased/decreased along with per-entry *trust degree*s. Prefetch is enabled only if the *stream trust degree* of the current CPU reaches its maximum value (i.e., 7). Removing *trust degree* reduces the hit rate of CBP from 98.6% to 60.0% during WebVMs batch replay (the details of the workload are presented in §5.3), which shows the effectiveness of *trust degrees*.

Prefetch-Current (PC). As Figure 4(b) shows, the simplest idea is to prefetch the stored block that is possible to be compared with the current input block (based on the stored hints) before any deduplication calculations, and then follow the basic deduplication logic to deduplicate the input block. In this way, fingerprint calculations and index looking up (f in Figure 4) are executed parallelly with NVM reads, and then part of the NVM read latency can be hidden.

Speculation (SP). Furthermore, as Figure 4(c) shows, we leverage hints to skip the fingerprint calculations and indexing by directly locating the deduplication metadata entry (see **Speculation* arrow in Figure 2). If the content-comparison demonstrates that the compared two blocks are the same, then the duplicate block is found; otherwise, we fall back to the basic deduplication logic. *SP* outperforms *PC* since *SP* can reduce (skip) many deduplication calculations.

Prefetch-Next (PN). More aggressively, as Figure 4(d) shows, to maximize parallelism, we utilize the stored speculation hints to suggest the block that is likely to be compared with the subsequent input block, and then we prefetch that block after loading the current to-be-compared block into CPU caches (2 in Figure 4). Therefore, the NVM read of the next to-be-compared can be parallel with the following CPU tasks (e.g., content-comparison). Now, NVM read is almost fully parallel with the CPU computations.

Cross-Block Prefetch (CBP). From the above explorations, we take *PN* as the fundamental of cross-block prefetch. However, we find that *PN* significantly degrades the deduplication performance at a high concurrency level (as shown in Figure 11 later in §5.4) since the large amount of extra prefetch I/O exacerbates the contention of NVM read buffer. Thus, we further introduce *Transition* technique to mitigate the issue by dynamically enabling/disabling prefetch according to concurrency level. Specifically, we maintain the number of threads that access NVM concurrently with an atomic variable and do not prefetch the next block if the number of threads reaches the specified threshold. Note that the threshold is a kernel module parameter. It is set to 6 by default (and we use this default value in our tests) because according to Figure 11, PN's throughput drops below SP's throughput when the number of threads ≥ 6 due to buffer contentions. Now, we obtain the final version of *CBP* (i.e., *PN*+*Transition*).

4.3.3 Speculative Prefetch: Put IBP and CBP Together

Generally, speculative prefetch enables the asynchronous NVM reads for content-comparison at both byte and block levels. Among them, CBP is frequently triggered when the workload exhibits good duplication continuousness (i.e., most hints are trusted according to trust degrees). Otherwise, Light-Dedup falls back to IBP when the duplication continuousness is poor (since most hints are not trusted). Therefore, the functionalities of CBP and IBP are complementary and are combined together to deliver fast content-comparison performance in both cases. The performance evaluation shows that speculative prefetch can achieve up to 118% performance improvement. More details can be obtained in §5.4.

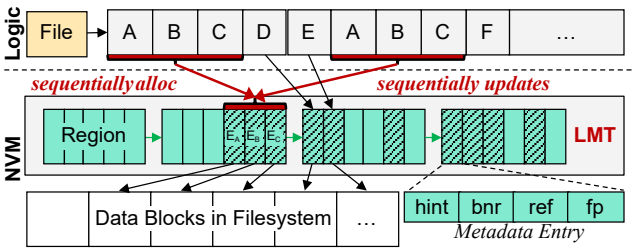


Figure 5: Illustration of Light-Meta-Table (LMT).

4.4 LMT: In-NVM Dedup Metadata Layout

In this subsection, we present the design of LMT, as illustrated in Figure 5. LMT is used for maintaining the mappings from fingerprints to physical blocks and providing hints for speculative prefetch. Furthermore, the deduplication metadata in LMT are laid out in the *region* granularity to reduce the read/write amplification of deduplication metadata access in NVM (i.e., allocating entries almost sequentially).

Deduplication Metadata Entry. In the LMT, each entry consists of 8 bytes *blocknr* (block number), 8 bytes *fp* (fingerprint), 8 bytes *refcnt* (reference count), and 8 bytes *hint* (used for speculative prefetch). The *blocknr* field refers to the corresponding block number in the file system, while the *refcnt* field refers to the number of references on a data block. The *fp* field stores the 8 bytes xxHash as the block's fingerprint. And the *hint* field contains 61 bits for the location of the subsequent speculated block's entry and 3 bits for *trust degree*.

Region-based Layout. To maintain the locality of deduplication metadata, we group metadata entries into *region*s and allocate entries in the currently used *region* almost sequentially. We use an in-DRAM variable ***Cur Region*** to represent this in-NVM region. For brevity, we do not distinguish *Cur Region* from the currently used in-NVM region. Each *region* is 4 KiB (aligned to the block size) so that the *region*s can be allocated by NVM file systems' block allocator directly [17, 66]. The *region*s are linked by 8 bytes pointers at the end of each *region*, and the first a few *region*s (*Region Header*) are reserved in a fixed place in NVM as the list head. In this way, the deduplication metadata table can grow dynamically. Therefore, Light-Dedup avoids unnecessary storage consumption and can scale flexibly when the storage size changes.

We regard a *region* as allocatable (i.e., we can use the *region* to allocate entries) if no more than half of the metadata entries (i.e., $4\,\text{KiB}/32\,\text{Byte}/2 = 64$ entries) are used in that region. Such design is a tradeoff between maintaining the locality of metadata entries' allocation and the space utilization of the region. To allocate regions in constant time, Light-Dedup maintains the positions of allocatable regions with the *Region Queue* in DRAM and keeps track of the number of valid entries in each region with an XArray [60]. Figure 6 illustrates the allocation and deletion of metadata entries:

- **Entry Allocation**. Light-Dedup checks the entries in the *Cur Region* one by one (①) until a free one is found (②), and then returns its position to the caller (③). If there is no free entry found (④), Light-Dedup evicts the *Cur Region* (⑤) and checks if the *Region Queue* is empty. If not, Light-Dedup takes out an allocatable region from the *Region Queue* as the new *Cur Region* (⑥(1)). Otherwise, Light-Dedup leverages the file systems' block allocator to allocate a new *region* as the *Cur Region*, and then links it into the tail of the in-NVM *region* list (⑥(2)). For the new *Cur Region*, Light-Dedup checks the entries in it one by one until a free entry is found (⑦) and returns its position to the caller (⑧).
- **Entry Deletion**. Light-Dedup first sets the *blocknr* of the target entry (⑨) to zero. If exactly half of the metadata entries in that *region* are free, then we insert this *region* back to the *Region Queue* to make it allocatable again (⑩). For the simplicity of concurrent control, the *region* will not be returned to the block allocator.

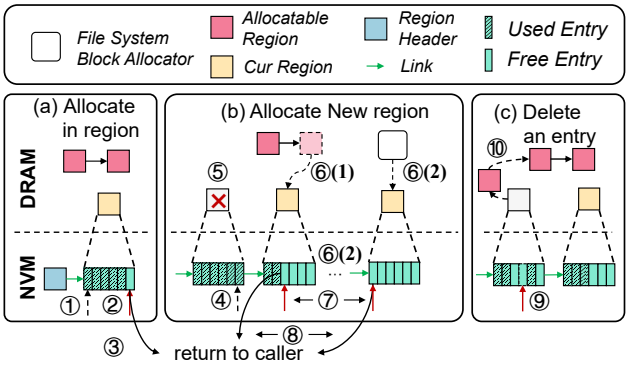


Figure 6: Illustration of region-based entry management. The cross-mark indicates that the *Cur Region* is evicted.

Reduction of Metadata I/O Amplification. To show this, we make the following analysis: (1) **For unique writes**, Light-Dedup allocates a metadata entry for each write almost sequentially in the *Cur Region*, so that the writes to these entries usually hit the NVM write buffer, and thus the metadata write amplification is reduced. (2) **For duplicate writes**, we assume the redundant data tend to be clustered, such as using *cp* or *rsync* to copy a file multiple times. In that case, since the file's deduplication metadata is allocated almost sequentially in a region, the subsequent accesses/modifications to them are also almost sequential, and thus the metadata read and write amplification are both reduced. The extensive study in §5.5 validates our analysis.

Avoidance of Garbage Collection (GC). Maintaining sequentiality (e.g., log-structured layout) often requires GC [66]. However, GC is complex and time-consuming. To address the issue, Light-Dedup does not reclaim allocated regions and allows to reuse them when half of the entries are free. In other words, Light-Dedup trades more NVM space for GC-free design. Such design does not bring significant storage consumption: assuming the capacity of NVM is x, then there are at most $\frac{x}{4\text{KiB}}$ unique blocks to be referenced by Light-Dedup.

Thus, at most $\frac{2x}{4\text{KiB}}$ entries are needed for GC-free design (i.e., all the allocated regions are half-full). Since each entry is 32 bytes, at most $\frac{2x}{4\text{KiB}} \times 32\,\text{Byte}/x \approx 1.56\%$ space of NVM is needed. Note that this is the worst case. The experiment shows that writing a 128 GiB non-duplicate file allocates 1.008 GiB of regions, which is only 0.79% of the data size.

4.5 Crash Consistency and Recovery

Light-Dedup maintains crash consistency lazily by collaborating with NVM file systems' recovery process. The lazy strategy guarantees crash consistency and avoids eager consistency overheads [9, 47, 68].

Normal Recovery: *Storing in-DRAM index, allocator, etc. to NVM and Reloading them back*. During the clean unmount, Light-Dedup stores in-DRAM *rhashtable* items and the valid entry counts in the reserved area in NVM. During the subsequent remount, Light-Dedup first initializes an empty index, and then inserts the saved items into it. Next, the valid entry counts are loaded into DRAM directly and the *Region Queue* is rebuilt accordingly. After this process, Light-Dedup is ready to accept new I/O requests.

Failure Recovery: *Fixing inconsistency of deduplication metadata in NVM and Reconstructing in-DRAM data structures*. To recover to the normal state, Light-Dedup scans the deduplication metadata during the file systems' recovery to fix two inconsistent cases: (1) A block is only referenced by the file system. It is a non-dedup block, so Light-Dedup does not reinsert it into the deduplication metadata table. (2) A block is only referenced by the deduplication metadata table. Since Light-Dedup treats the file system's metadata as the true source of information, it invalidates the corresponding deduplication metadata entry. After this, Light-Dedup rebuilds its in-DRAM structures similar to the normal recovery.

4.6 Portability

Port to Future NVM Devices. Although Optane DCPMM exited the market recently, we are still confident about NVM technology because it bridges the performance gap between DRAM and SSDs. Our work focuses on the common features of storage-type NVMs as discussed in §2.1, e.g., long media read latency, memory interface, and coarse media access granularity. Therefore, we believe our work can be applied to future commercial storage-type NVMs.

Port to Future CXL-based Devices. Compute Express Link (CXL) [15, 25, 53] is an emerging interconnect standard. We believe that Light-Dedup can also be applied to the storage systems using CXL (e.g., NVMs interconnected with CXL) if the systems exhibit the common features that Light-Dedup focuses on. We leave this as our future work.

Port to Other Instruction Sets. Although Light-Dedup is currently implemented on x86, the idea of hiding long NVM media read latency with speculative prefetch can be applied to other instruction sets with prefetch instructions, such as ARM with the PRFM instruction [2].

5 Performance Evaluation

This section seeks to answer the following questions: (*i*) How does Light-Dedup perform against state-of-the-art NVM (deduplication) file systems? (§5.2) (*ii*) How does Light-Dedup perform in real-world scenarios? (§5.3) (*iii*) How does speculative prefetch in LRBI contribute to final performance? (§5.4) (*iv*) How efficient is the design of LMT? (§5.5) (*v*) How expensive is the Light-Dedup recovery mechanism? (§5.6)

5.1 Experimental Setup

Testbed. We evaluate Light-Dedup on a server with an Intel Xeon Gold 5218 CPU clocked at 2.3 GHz, which has 16 cores (32 hyper-threads) and 22 MiB of L3 cache with *clwb* support. The machine is equipped with 512 GiB Optane DCPMM (2×256 GiB DIMMs) in non-interleaved AppDirect Mode, and 128 GiB DRAM (4 × 32 GiB DIMMs). The server runs CentOS with kernel 5.1.0 modified by NOVA [48].

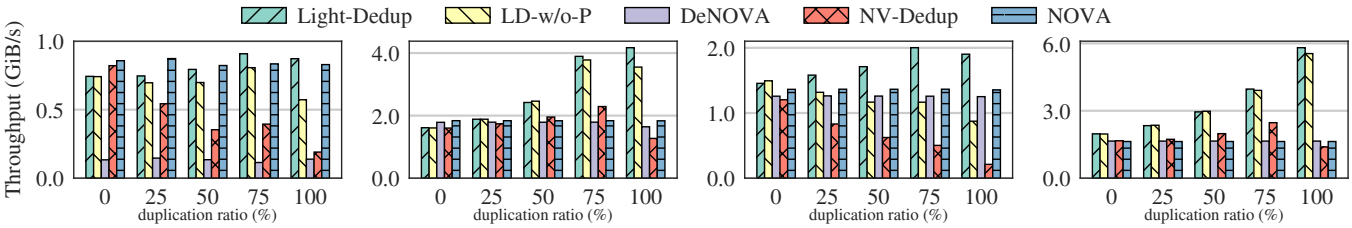
Compared Systems. We compare Light-Dedup with NOVA, NV-Dedup, DeNOVA, and LD-w/o-P. Among them, the source code of NV-Dedup is not publicly available, thus we re-implement it on top of NOVA[3], following the same configurations in their paper [58]. For DeNOVA, we implement the Deduplication Daemon (DD) based on the open-source version and deduplicate the data in the background aggressively (i.e., *DeNOVA-Immediate* [28]).

Methodology. FIO [4] is used to measure extensive I/O performance. We use *sync* as I/O engine to guarantee the persistence; 0–75% duplication ratio is emulated with parameter *dedupe_percentage*. For the 100% duplication ratio, we perform the same FIO twice and measure the performance of the second run. The reason is that 100% *dedupe_percentage* results in issuing a few unique blocks, which is quite different from the real scenarios. Moreover, we also measure four real-world workloads: copying compiled Linux kernel as code archiving scenario, replaying three realistic traces as frequent data operations scenarios. Among these traces, *WebVMs* and *Mails* are collected from FIU [27, 33]. *Homes* is generated from 50 students' home directories on our OS Lab server: we break the files into 4 KiB blocks, generate each of them md5 digest similar to FIU traces, and use the files' creation time as timestamps. Each measurement is repeated 5 times, and the average values are presented. All coefficients of variation are less than 5%, which suggests reproducibility and stability. The evaluation scripts are available at https://github.com/Light-Dedup/tests.

5.2 Microbenchmarks

We use FIO to evaluate the write performance of Light-Dedup under the different duplication ratios, write patterns, and concurrency levels, i.e., a single thread and 8 threads. Adding more threads does not contribute to the performance improvement due to the contention on the Optane DCPMM [67]. The workload data size is set to 128 GiB to observe a more stable result. Note that under the 100% duplication ratio, each

[3] https://github.com/Light-Dedup/nv-dedup



(a) Single thread (4KiB per I/O) (b) 8 threads (4KiB per I/O) (c) Single thread (2MiB per I/O) (d) 8 threads (2MiB per I/O)

Figure 7: Microbenchmark with FIO under different write patterns (4 KiB/2 MiB per I/O) and concurrency levels.

run performs 64 GiB writes. Further, to better support the large workload in FIO, we modify NOVA by removing the threshold of log extending and by making the extension always double the size of the inode log, which eliminates a substantial amount of many unnecessary *Fast GC*s [66]. The experimental results are shown in Figures 7.

Block-based I/O (4 KiB). Figures 7(a) and (b) represent the throughput of 4 KiB I/O writes. We find that: (1) Light-Dedup achieves 1.70–4.58× throughput over NV-Dedup when the duplication ratio is ≥75% since NV-Dedup suffers from cryptographic hash calculation overheads, while LRBI enables the efficient deduplication in either concurrency levels. (2) Light-Dedup is 3–15% slower than NOVA for the 0% duplication ratio, but the throughput can be 1.05–2.28× to the throughput of NOVA when the duplication ratio is ≥75%. (3) *IBP* contributes to 1–52% performance improvement under single thread compared to LD-w/o-P. However, its contribution is reduced with the increasing threads number due to dramatically enlarged read/write asymmetry; thus, the improvement of *IBP* is diluted. Notably, *CBP* cannot work ideally across syscalls. A possible reason is that the load queue [49] is flushed during the context switch.

Continuous Block I/O (2 MiB). Figures 7(c) and (d) represent the throughput of 2 MiB I/O writes, i.e., writing 512 4 KiB blocks in one syscall. We find that: (1) all the evaluated file systems gain higher write performance due to fewer syscalls. Among them, DeNOVA benefits the most since the contention of DD's dequeue and enqueue decreases significantly. (2) Under the single thread, Light-Dedup achieves 72–118% performance improvement when the duplication ratio is ≥75% compared to LD-w/o-P since *Cross-Block Prefetch* can leverage the locality of workloads to speculate the subsequent block efficiently in a single syscall.

Read-Write Interference. Although Light-Dedup trades slow duplicate writes for faster asynchronous reads, the mixed read/write I/O under multi-thread environments can potentially interfere with each other. This is because writing duplicate blocks require content comparisons and do not write redundant data, which can be seen as a reader-like operation. Conversely, threads that write unique blocks can be seen as writer-like because they aim to write new blocks. Such interference has been previously observed in NyxCache [61] and MT [69]. However, the goal of Light-Dedup is to improve overall performance by eliminating duplicate data blocks, and

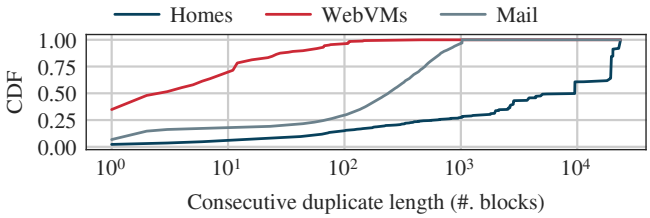


Figure 8: Cumulative distribution function of duplication continuousness of evaluated traces.

Table 3: Characteristics of evaluated real-world workloads.

Workload	Total I/O	Write Prop.	Dup Ratio	Granularity
Copy	13.85 GiB	100%	100%	2 MiB
Homes	63.52 GiB	100%	84%	4 KiB for Blk Max 2 MiB for Bat
WebVMs	54.53 GiB	78%	47%	4 KiB for Blk Max 2 MiB for Bat
Mails	173.27 GiB	91%	95%	4 KiB for Blk Max 2 MiB for Bat

we argue that even though individual tasks can be negatively affected, the overall performance can still be improved.

To show this, we run a set of experiments (not shown in the figure) on the case of 50% duplication ratio and 8 threads. To obtain the separate bandwidth of readers and writers, we make four threads write the duplicated data (as readers) while the remaining four write unique data (as writers). We observe that the bandwidth of Light-Dedup decreases to 1316 MiB/s for readers and 1052 MiB/s for writers when compared to the bandwidth of non-interfered systems with 4 readers (3380 MiB/s) and 4 writers (1608 MiB/s), respectively. However, when compared to the bandwidth of non-interfered systems with 8 writers (1624 MiB/s), the overall bandwidth of Light-Dedup with mixed 4 readers and 4 writers is improved to 2368 MiB/s. Experimental results show that Light-Dedup's non-cryptographic-hash-based deduplication approach can improve overall deduplication performance even in the presence of read-write interference.

5.3 Real-world Scenarios

In this subsection, we study the performance of Light-Dedup under real-world scenarios. The characteristics of the four workloads are summarized in Table 3. Specifically, **for *Copy***, we copy compiled Linux kernel twice from SSD to NVM, and the bandwidth of the second copying is measured. *Copy* can be considered as a real-world application that uses NVM as the storage of code repositories. **For the other three traces**,

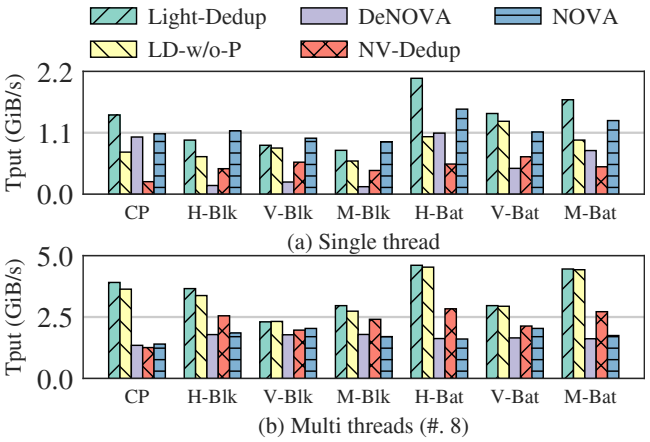


Figure 9: Performance comparison of real-world scenarios. Here, H, V, M indicate *Homes*, *WebVMs*, and *Mails*, respectively. Suffix "-Blk" (***block replay***) and "-Bat" (***batch replay***) indicate that each I/O processes one block and a batch of no more than 512 consecutive blocks, respectively.

we implement *trace-replayer*[4] to emulate the behaviors of a real-world application and replay traces. Unlike previous block-level trace replay tools [27, 33], *trace-replayer* can batch consecutive logical blocks in one syscall, which has the same rationale as many real-world applications (e.g., batching the reads/writes using a local buffer). The insights can be reflected in Figure 8. *Homes* and *Mails* traces show good spatial locality, which the speculative prefetch can well exploit. For *WebVMs*, there are only 30% duplicate blocks whose consecutive length is $>$10 blocks (point $(10, 0.70)$), which shows relatively poor continuousness.

Figure 9 compares the throughput of Light-Dedup and the other four approaches under different real-world workloads. For DeNOVA, we configure *trace-replayer* to replay the traces by appending data but ignoring the given data offset due to its bugs of handling overlapping writes. Light-Dedup shows the best deduplication performance under all workloads in most cases. In particular, (1) during *Copy*, Light-Dedup is much faster than other approaches since speculative prefetch markedly hides read latency under a single thread (due to the continuousness of the workload). (2) During *block reply*, *In-Block Prefetch* shows its effectiveness under a single thread. With the increasing number of threads and the enlarged read/write asymmetry, the throughput of LD-w/o-P catches up with that of Light-Dedup. (3) During *batch replay*, speculative prefetch contributes a lot to single-thread performance with a high duplication ratio. For example, for single-thread *Mails*, Light-Dedup achieves 1.28× write throughput compared to NOVA under a single thread. However, LD-w/o-P cannot even catch up with NOVA's throughput.

5.4 Speculative Prefetch Efficiency

In this subsection, we study the efficiency of prefetching and speculation by using FIO with block size set to 2 MiB, and *Copy* as our benchmarks.

Single-thread Comparison. Figure 10(a) writes the same 64 GiB data twice (using **FIO**) with a single thread, and the second writing bandwidth is measured. *PN* significantly exceeds other variants by 1.11–2.19×, mainly because prefetch enables the parallelism of CPU calculation and NVM I/O and significantly reduces content-comparison time. We further measure the deduplication performance of *Copy* under a single thread. Figure 10(b) presents the throughput of the second copy. The result is similar to FIO, except the overall throughput is lower. This is because there are 44% small files (less than 4 KiB) in the Linux kernel source code, and these small files degrade the deduplication performance.

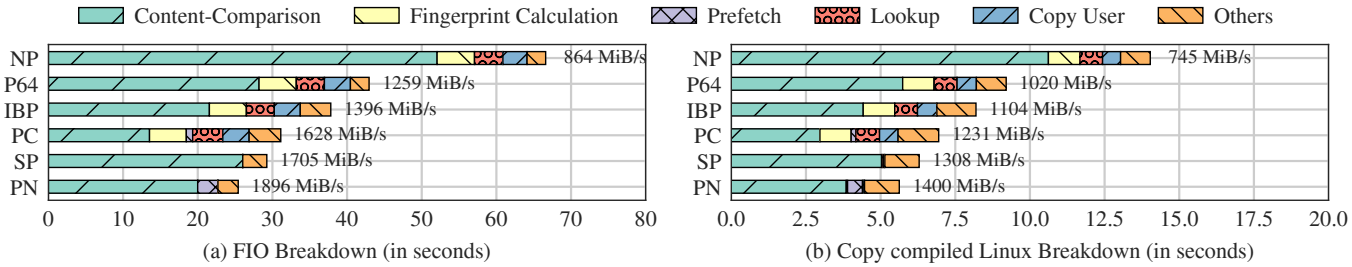
Multi-thread Comparison and Observations. To study how *PN* scales with the increasing concurrency level, we use the same FIO benchmark but with the number of threads set to 1–16. Figure 11 presents the second write performance. The figure shows that: (1) The throughput of *PN* is 1.03–1.29× and 1.51–2.19× that of *SP* and LD-w/o-P when the threads number ≤ 5 since *PN* prefetches NVM data into the CPU cache, which reduces content-comparison time. (2) When the threads number ≥ 6, *SP* shows about 1.11–1.80× throughput compared to *PN*. According to the breakdown performance (not shown in the figure due to space limits), we find that the content-comparison time of *PN* rises up to 1.65× to that of *SP* under 16 threads, which suggests the large amount of extra prefetch I/O exacerbates the contention of in-NVM buffers and thus leads to longer I/O latency. (3) The evaluation shows that *CBP* (i.e., *PN+Transition*, see §4.3) combines the benefits of *SP* and *PN* and scales well with the increment of threads.

5.5 Metadata I/O Amplification in LMT

To study the efficiency of region-based layout in LMT, we have designed a workload to quickly age the file system. (1) We first write a 128 GiB file (2 MiB per I/O) to a newly mounted deduplication file system (Fresh System). (2) Next, we punch the file randomly by using `fallocate()` until the file size is reduced to half. This step creates random holes in the file system, which emulates the aging process (Aged System). (3) Finally, we write another 64 GiB file to fill the holes. In the aged system, the spatial distribution of free entries is random. Inappropriate metadata management (e.g., *entry-based* layout used in NV-Dedup [58]) can cause severe read/write amplification and consequently decline the system's I/O performance.

Table 4 shows the comparison between *region-based* and *entry-based* metadata layout in multiple dimensions under the aging workload. The evaluation shows that *region-based* consistently outperforms *entry-based* in all the dimensions and can resist fragmentation problems. Maintaining the locality of entries significantly reduces the write amplification under the aged system (i.e., from *entry-based*'s 9.86× to *region-based*'s 3.43×), and improves about 11.6% write throughput relative to *entry-based*. The results suggest that *region-based* meta-

[4]The tool is available at https://github.com/Light-Dedup/nvm_tools



(a) FIO Breakdown (in seconds)

(b) Copy compiled Linux Breakdown (in seconds)

Figure 10: Performance comparison and breakdowns of different variants. For simplicity, we denote LD-w/o-P as *NP* (i.e., no speculative prefetch), and recall that IBP is *In-Block Prefetch*. Note that *Cross-Block Prefetch (CBP)* is effectively equivalent to *PN* in single-thread experiments; thus, it is omitted in the above figures.

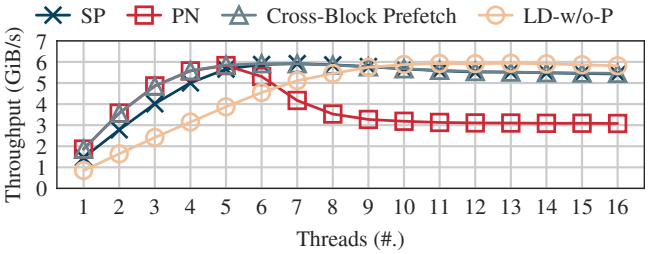


Figure 11: Performance comparison between different deduplication strategies under different threads.

Table 4: Comparison of region-based and entry-based metadata layout under the aging workload.

Dimension	Fresh System (128 GiB)		Aged System (64 GiB)	
	Region	Entry	**Region**	Entry
Reads Per Block (B)	116.28 (2.91×)	126.94 (3.17×)	**244.19 (6.1×)**	774.13 (19.35×)
Writes Per Block (B)	75.75 (1.89×)	79.56 (1.99×)	**137.17 (3.43×)**	394.54 (9.86×)
Throughput (MiB/s)	1747.5	1690.6	**1336.72**	1197.76

data layout is more friendly to NVM deduplication, especially in an aged file system (which is common in the production environments).

5.6 Recovery Overheads

Table 5 studies the unmount time, normal recovery time, and failure recovery time of NOVA and Light-Dedup with different sizes of files (the total number of files is fixed to 32). The results show that although Light-Dedup causes overheads during recovery, its unmount and failure recovery time remains the same trend as NOVA (linearly) as the file size grows. We argue that trading longer unmount/recovery time for more efficient runtime is reasonable for NVM deduplication.

6 Discussion

Memory Consumption of *rhashtable*. We observe that the memory consumption of writing 128 GiB data under 0%, 25%, 50%, and 75% duplication ratio is 1.26 GiB, 1.08 GiB, 658 MiB, and 331 MiB, respectively. The experimental results indicate that the memory consumption is less than 1% of the data size (e.g., $1.26GiB/128GiB \approx 0.98\%$).

Hardware-based Cryptographic Hash Calculation. There are several hardware accelerators developed for efficiently calculating cryptographic hash [5, 14, 42, 55]. However, they are not widely deployed and require special hardware. Therefore, they are not considered in this paper.

Table 5: Comparison of recovery overheads.

Dimension	File system	File system utilization (GiB)		
		32 × 1	32 × 2	32 × 4
Umount Time (s)	NOVA	0.385	0.775	1.502
	Light-Dedup	0.551	1.095	2.099
Normal Recovery Time (s)	NOVA	0.015	0.015	0.015
	Light-Dedup	0.617	1.223	2.398
Failure Recovery Time (s)	NOVA	0.315	0.488	0.829
	Light-Dedup	1.260	2.372	4.604

Scalability on Multiple Optane DCPMMs. We run a 32 GiB FIO workload with 75% duplication ratio on two interleaved 256 GiB DCPMMs. The experimental results show that the throughput of Light-Dedup increases from 952 MiB/s to 6238 MiB/s with 1–16 threads, suggesting Light-Dedup can scale with increasing threads on multiple Optane DCPMMs.

7 Conclusion and Future Work

In this paper, we propose Light-Dedup, a light-weight inline deduplication framework for NVM file systems. With the NVM-aware Light-Redundant-Block-Identifier (LRBI) and Light-Meta-Table (LMT), Light-Dedup is able to maximize NVM deduplication performance by fully considering NVM's I/O mechanisms (e.g., long media read latency). Evaluation results show that the deduplication cost is low, and the performance can be enhanced if the duplication ratio is high. Since memory usage is sensitive to server environment [28], we plan to incorporate other memory-efficient hash table design [36, 45, 73, 74] to optimize Light-Dedup's index further.

Acknowledgments

We thank our shepherd, Youjip Won, and the anonymous reviewers for their constructive comments and insightful suggestions. This research was partly supported by the National Natural Science Foundation of China under Grant no. 61972441 and U22B2022; Shenzhen Science and Technology Innovation Program under Grant no. RCYX20210609104510007; Guangdong Basic and Applied Basic Research Foundation under Grant 2021A1515012634.

References

[1] Mijin An, Soojun Im, Dawoon Jung, and Sang-Won Lee. Your read is our priority in flash storage. *Proceedings of the VLDB Endowment*, 15(9):1911–1923, 2022.

[2] ARM. Arm cortex-a75 core technical reference manual r2p0, 2023. https://developer.arm.com/documentation/100403/0200/functional-description/level-1-memory-system/data-prefetching.

[3] Amro Awad, Sergey Blagodurov, and Yan Solihin. Write-aware management of NVM-based memory extensions. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2925426.2926284.

[4] Jens Axboe. Flexible i/o tester, 2017. https://github.com/axboe/fio.

[5] Sergei Brazhnikov. A hardware implementation of the SHA2 hash algorithms using CMOS 28nm technology. In *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 1784–1786. IEEE, 2020.

[6] Mu-Tien Chang, Paul Rosenfeld, Shih-Lien Lu, and Bruce Jacob. Technology comparison for large last-level caches (l 3 cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM. In *2013 IEEE 19th international symposium on high performance computer architecture (HPCA)*, pages 143–154. IEEE, 2013.

[7] Wande Chen, Zhenke Chen, Dingding Li, Hai Liu, and Yong Tang. Low-overhead inline deduplication for persistent memory. *Transactions on Emerging Telecommunications Technologies*, page e4079, 2020.

[8] Youmin Chen, Youyou Lu, Bohong Zhu, et al. Scalable persistent memory file system with kernel-userspace collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 81–95, 2021.

[9] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Consistency without ordering. In *FAST*, page 9, 2012.

[10] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1683–1694, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Yann Collet. xxhash: Extremely fast hash algorithm, 2016. https://github.com/Cyan4973/xxHash.

[12] 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 ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[13] Jonathan Corbet. Relativistic hash tables, part 1: Algorithms, 2014. https://lwn.net/Articles/612021/.

[14] Luigi Dadda, Marco Macchetti, and Jeff Owen. The design of a high speed asic unit for the hash function sha-256 (384, 512). In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 3, pages 70–75. IEEE, 2004.

[15] Debendra Das Sharma. Keynote 1: Compute express link (cxl) changing the game for cloud computing. In *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages xii–xii, 2021.

[16] Biplob K Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX annual technical conference (USENIX ATC'10)*, pages 1–16, 2010.

[17] 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 Ninth European Conference on Computer Systems (EuroSys'14)*, pages 1–15, 2014.

[18] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):855–868, 2016.

[19] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 181–192, Philadelphia, PA, June 2014. USENIX Association.

[20] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 331–344, Santa Clara, CA, February 2015. USENIX Association.

[21] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Energy-efficient hybrid DRAM/NVM main memory. In *Proceedsings of 2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 492–493, 2015.

[22] Intel. ipmctl, 2018. https://github.com/intel/ipmctl.

[23] Intel. Intel® 64 and ia-32 architectures software developer manuals, 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[24] Joseph Izraelevitz, Jian Yang, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[25] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 45–51, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Myoungsoo Jung, John Shalf, and Mahmut Kandemir. Design of a large-scale storage-class RRAM system. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 103–114, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[28] Hyungjoon Kwon, Yonghyeon Cho, et al. Denova: Deduplication extended nova file system. In *IPDPS*, pages 1–12. IEEE, 2022.

[29] Giusy Lama. *Phase Change Memory (PCM) for High Density Storage Class Memory (SCM) Applications*. PhD thesis, Université Grenoble Alpes [2020-], 2022.

[30] Hyokeun Lee, Moonsoo Kim, Hyunchul Kim, Hyun Kim, and Hyuk-Jae Lee. Integration and boost of a read-modify-write module in phase change memory system. *IEEE Transactions on Computers*, 68(12):1772–1784, 2019.

[31] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *Asian Symposium on Programming Languages and Systems*, pages 244–265. Springer, 2019.

[32] Jingwei Li, Zuoru Yang, et al. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, pages 1–15, 2020.

[33] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, February 2016. USENIX Association.

[34] Yu-Pei Liang, Tseng-Yi Chen, Yuan-Hao Chang, Shuo-Han Chen, Pei-Yu Chen, and Wei-Kuan Shih. Rethinking last-level-cache write-back strategy for mlc stt-ram main memory with asymmetric write energy. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2019.

[35] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. In *Proceedings of the VLDB Endowment*, page 1147–1161, April 2020.

[37] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. Pod: Performance oriented i/o deduplication for primary storage systems in the cloud. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 767–776. IEEE, 2014.

[38] Paul McKenney. What is rcu, fundamentally?, 2007. https://lwn.net/Articles/262464/.

[39] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[40] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[41] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. *Supercomputing frontiers and innovations*, 1(3):19–55, 2014.

[42] Rahul P Naik and Nicolas T Courtois. Optimising the SHA256 hashing algorithm for faster and more efficient bitcoin mining. *MSc Information Security Department of Computer Science UCL*, pages 1–65, 2013.

[43] Prashant J Nair, Chiachen Chou, Bipin Rajendran, and Moinuddin K Qureshi. Reducing read latency of phase change memory via early read and turbo read. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 309–319. IEEE, 2015.

[44] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. *ACM SIGPLAN Notices*, 52(4):135–148, 2017.

[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 Technologies (FAST'19)*, pages 31–44, 2019.

[46] Heba Nashaat, Nesma Ashry, and Rawya Rizk. Smart elastic scheduling algorithm for virtual machine migration in cloud computing. *The Journal of Supercomputing*, 75(7):3842–3865, 2019.

[47] Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P. C. Lee, and John C. S. Lui. Live deduplication storage of virtual machine images in an open-source cloud. In *Proceedings of Middleware 2011*, pages 81–100, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[48] NVSL. Nova: Non-volatile memory accelerated log-structured file system, 2017. https://github.com/NVSL/linux-nova.

[49] Il Park, Chong Liang Ooi, and TN Vijaykumar. Reducing design complexity of the load/store queue. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 411–422. IEEE, 2003.

[50] Moinuddin K Qureshi, Michele M Franceschini, and Luis A Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–11. IEEE, 2010.

[51] Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-Montaño. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–11, 2010.

[52] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020.

[53] Debendra Das Sharma. Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 5–12, 2022.

[54] ChunYi Su, David Roberts, Edgar A León, Kirk W Cameron, Bronis R de Supinski, Gabriel H Loh, and Dimitrios S Nikolopoulos. Hpmc: An energy-aware management system of multi-level memory architectures. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 167–178, 2015.

[55] Vikram Suresh, Sudhir Satpathy, Sanu Mathew, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, and Ram Krishnamurthy. A 230mv-950mv 2.8 tbps/w unified sha256/sm3 secure hashing hardware accelerator in 14nm tri-gate cmos. In *ESSCIRC 2018-IEEE 44th European Solid State Circuits Conference (ESSCIRC)*, pages 98–101. IEEE, 2018.

[56] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *Proceedings of the 2014 Ottawa Linux Symposium (OLS'14)*, pages 83–95. Citeseer, 2014.

[57] K. Venkatesh and D. Narasimhan. Revealing the novel precise subset identification and deduplication of audio substance over the shared public environment. *The Journal of Supercomputing*, Feb 2022.

[58] Chundong Wang, Qingsong Wei, Jun Yang, Cheng Chen, Yechao Yang, and Mingdi Xue. Nv-dedup: High-performance inline deduplication for non-volatile memory. *IEEE Transactions on Computers*, 67(5):658–671, 2017.

[59] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE 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.

[60] Matthew Wilcox. Xarray. https://www.kernel.org/doc/html/v5.1/core-api/xarray.html.

[61] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, Santa Clara, CA, February 2022. USENIX Association.

[62] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[63] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX annual technical conference (USENIX ATC'11)*, pages 26–30, 2011.

[64] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Cong Xu, Xiangyu Dong, Norman P Jouppi, and Yuan Xie. Design implications of memristor-based RRAM cross-point structures. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[66] 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'16)*, pages 323–338, 2016.

[67] 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'20)*, pages 169–182, 2020.

[68] Qirui Yang, Runyu Jin, and Ming Zhao. Smartdedup: optimizing deduplication for resource-constrained devices. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 633–646, 2019.

[69] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT^2: Memory bandwidth regulation on hybrid nvm/dram platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216, 2022.

[70] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. Htmfs: Strong consistency comes for free with hardware transactional memory in persistent memory file systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 17–34, 2022.

[71] Jianhui Yue and Yifeng Zhu. Accelerating write by exploiting PCM asymmetries. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 282–293, 2013.

[72] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 269–282, 2008.

[73] Xiaomin Zou, Fang Wang, Dan Feng, Chaojie Liu, Fan Li, and Nan Su. Hmeh: write-optimal extendible hashing for hybrid DRAM-NVM memory. In *2020 36th Symposium on Mass Storage Systems and Technologies (MSST)*, 2020.

[74] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.

[75] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2018)*, pages 442–454. IEEE, 2018.

TiDedup: A New Distributed Deduplication Architecture for Ceph

Myoungwon Oh[1] Sungmin Lee[1] Samuel Just[2] Young Jin Yu[1] Duck-Ho Bae[1]
Sage Weil[3] Sangyeun Cho[1] Heon Y. Yeom[4]
[1]*Samsung Electronics Co.* [2]*IBM* [3]*Ceph Foundation* [4]*Seoul National University*

Abstract

This paper presents *TiDedup*, a new cluster-level deduplication architecture for Ceph, a widely deployed distributed storage system. Ceph introduced a cluster-level deduplication design before; unfortunately, a few shortcomings have made it hard to use in production: (1) Deduplication of unique data incurs excessive metadata consumption; (2) Its serialized tiering mechanism has detrimental effects on foreground I/Os, and by design, only provides fixed-sized chunking algorithms; and (3) The existing reference count mechanism resorts to inefficient full scan of entire objects, and does not work with Ceph's snapshot. *TiDedup* effectively overcomes these shortcomings by introducing three novel schemes: Selective cluster-level crawling, an event-driven tiering mechanism with content defined chunking, and a reference correction method using a shared reference back pointer. We have fully validated *TiDedup* and integrated it into the Ceph mainline, ready for evaluation and deployment in various experimental and production environments. Our evaluation results show that *TiDedup* achieves up to 34% data reduction on real-world workloads, and when compared with the existing deduplication design, improves foreground I/O throughput by 50% during deduplication, and significantly reduces the scan time for reference correction by more than 50%.

1 Introduction

Open source infrastructure management systems [18, 30] have helped cloud providers of different scales deliver services at low costs [4, 7, 14, 37]. Progresses in corporate digitalization and new classes of data-intensive applications are fueling fast growth of data in the cloud and call for scalable and efficient data storage [17, 34]. For such cloud serving storage, deduplication is expected to offer a means to mitigate the cost issue. However, most general-purpose distributed storage systems in use today do not provide cluster-level deduplication.

Several technical challenges partially explain why that is the case. First of all, all components in a successful deduplication architecture should be designed with scalability in mind. Moreover, the architecture should consider various data types, like file, block, and object, in search for data reduction opportunities. Last but not least, the architecture must have compatibility with existing services in the storage system, like snapshot operation. More comprehensive research is needed to make deduplication generally available in distributed storage systems (see Section 8 for further discussions).

The situation is no different in Ceph [42]. It has become a dominant open source distributed storage system in cloud environments thanks to its design that takes reliability and scalability as the first priority [7, 31]. However, capacity optimizations like deduplication were not seriously considered in early designs, and thus, Ceph has failed to meet growing data demands in various installed configurations.

Recently, a cluster-level deduplication design was proposed for Ceph [29]. While the work presents worthy efforts (and is a baseline in our evaluation in Section 5), it contains critical technical issues that we believe will hold back its use in production environments. We capture three of them here.

First, the prior design blindly performs deduplication regardless of the amount of unique contents in a target object. Deduplicating a unique object brings no benefit and only increases computation and storage overheads. Since the amount of unique contents in objects depends highly on the workload, this approach might undermine deduplication efficiency, even more so in Ceph because it is a general purpose storage system for diverse data sets.

Second, the prior design relies on a limited tiering architecture that depends on coarse-grained processing and fixed size chunking (FSC). A single background thread of Object Storage Daemon (OSD) is responsible for all tasks required for deduplication. This architecture suffers performance degradation due to time-consuming object enumeration with a lock and a limitation on on-demand deduplication by external agents. Moreover, FSC is not always the most effective way to find duplicate chunks in real-world workloads [28, 47].

Lastly, the prior design approach has drawbacks in its reference management method. Specifically, it does not work hand in hand with Ceph's snapshot feature because it lacks chunk reference management on an object snapshot. In addition, the reference counting method in the prior work takes a significant amount of time to identify reference mismatches on deduplicated chunks, because it requires full object search in the storage pool to count references.

In this work, we propose a new cluster-level deduplication

architecture, called *TiDedup*, which does not have the aforementioned issues while respecting the core design principles of Ceph. Thanks to its design choices and implementation strategies, *TiDedup* effectively targets a general purpose distributed storage system for both primary and backup storage [10, 11], by providing file, object, and block services. *TiDedup* incorporates three key design schemes:

1. Selective cluster-level crawling. *TiDedup* implements a crawler process that incrementally searches and identifies redundant chunks to selectively trigger deduplication. By doing so, *TiDedup* successfully removes only redundant chunks with minimum overheads.

2. Event-driven tiering mechanism with content defined chunking (CDC). *TiDedup* eliminates background work in OSD and is designed to execute reactions upon an event to handle multiple requests concurrently. On top of that, new tiering APIs (*set_chunk*, *tier_flush*, *tier_evict*, and *tier_promote*), as well as control and data path with CDC, are introduced.

3. Object ID (OID) shared reference scheme. We propose an efficient reference management method using OID. By sharing OID reference information between adjacent snapshots, *TiDedup* not only makes deduplicated objects compatible with snapshot, but also minimizes messages between OSDs at snapshot creation time. In addition, by using OID as a backpointer, *scrub*—a job to identify and fix inconsistencies (e.g., reference mismatch)—is able to check whether the reference is valid without performing a full object search.

Our evaluation results show that *TiDedup* effectively saves storage space by up to 34% on real-world workloads including industrial manufacture data and corporate cloud services. Moreover, *TiDedup* outperforms the prior approach by 50% in throughput during deduplication. Importantly, our implementation passes the *teuthology test* [38], Ceph's quality test suite, demonstrating the robustness and readiness of *TiDedup* for real-world evaluation and production uses. This paper makes the following contributions:

- We demonstrate the challenges in modern distributed storage system when applying deduplication and overcome the challenges by introducing *TiDedup* (Section 3 and 4). Compared to the previous approach [29], *TiDedup* is scalable and compatible with Ceph's existing design philosophy and features. *TiDedup* allows the use of various chunking algorithms with extensibility, different from other tiering-based architectures [9, 49]. Our cluster-level reference management design is more efficient than existing reference count techniques [6].
- We propose a pluggable design that is applicable to Ceph and show how the design works in detail (Section 4). Since our proposal is based on a hash algorithm to locate objects, it can be applied to other systems that build on a similar hash algorithm, like GlusterFS [15], Swift [36] and Cassandra [19].
- We perform comprehensive evaluation using a realistic experimental setup (Section 5). Our evaluation shows that *TiDedup* effectively performs deduplication while having little impact on foreground I/O, and achieves more than 50% scrub time reduction, compared to the existing approach. We discuss design trade-offs based on evaluation results (Section 6).
- *TiDedup* has been merged into the Ceph main branch and is the default deduplication engine for Ceph. *TiDedup* is available to anyone for evaluation and production.

2 Background

This section provides a brief overview of Ceph and explains several key terms that will be used throughout this paper.

2.1 Ceph

Ceph [42] is an industry-leading open-source distributed storage system. It provides file, object, and block services to clients on a unified distributed object store called a Reliable Autonomic Distributed Object Store (RADOS), comprised of multiple OSDs. Ceph exploits a decentralized hash algorithm, known as CRUSH [43], to determine the object location within RADOS. An OSD is responsible for serving, replication, and recovery of objects on top of a block device. In this paper, we mainly deal with RADOS because it is the primary component involved with all three services.

Ceph uses three terms—POOL (tier), Placement Group (PG), and object name—to represent the object location. The tier is a global namespace that consists of PGs, and PG indicates a logical group of objects. In addition, every PG links a replication group, which maps a set of OSDs. By using Ceph's OID format—this includes the tier and object name—as an input argument of CRUSH calculation, Ceph can find out PG ID. Therefore, if an OID is given, Ceph is able to look up the OSD where the object is located. Note that objects that belong to different PGs can co-exist in the same OSD.

2.2 Deduplication

Deduplication is a well-known data reduction technique to eliminate redundancy in data [24, 44]. Its typical process is composed of the following three steps: chunking (e.g., fixed size chunking [33] and content-defined chunking [26, 46]), fingerprinting (e.g., sha256), and comparison with existing fingerprints [29, 46]. However, performing deduplication does not always lead to space savings because datasets have different amount of redundancy; for example, datasets of web and mail servers [25] may have lower duplicate data than backup [39] and registry workloads [16, 49].

2.3 How deduplication works with Ceph

Previously, Ceph proposed selective post processing deduplication based on a tiering mechanism [29]. In this design, Ceph divides a logical storage space into two groups: a base tier and a chunk tier. They manage metadata objects and chunk objects, respectively.

Upon a write request, every object is written to the base tier first as a metadata object; at this time, it is considered

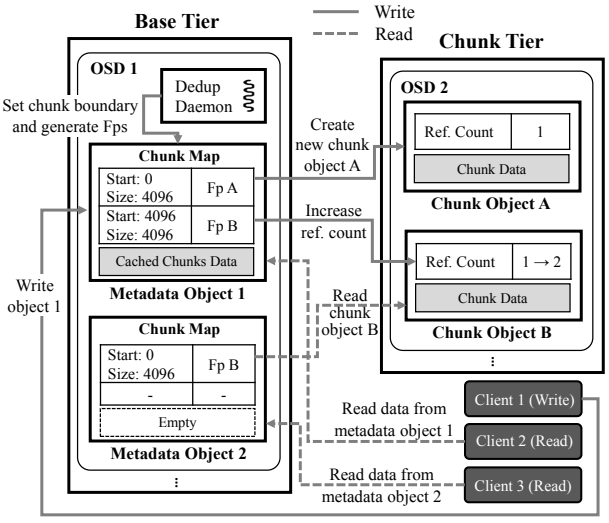


Figure 1: Ceph deduplication in prior work [29].

hot and is not eligible for deduplication. When the metadata object becomes cold (by LRU), OSD divides its content into several chunks using FSC, and generates a fingerprint from the corresponding chunk (e.g., Fp A and Fp B of Metadata Object 1 in Figure 1). Next, OSD stores the generated fingerprints in a chunk map of the metadata object. Then, using the fingerprint as OID, OSD stores the chunk as a chunk object in the chunk tier (Chunk Object A and B). If the chunk object already exists, OSD just increments the reference count by one. If not, OSD writes the chunk object, and sets the number of references to one.

To serve a read request, OSD searches metadata objects using the original OID. If the cached data resides in an object like Metadata Object 1 in Figure 1, the cached data is returned to Client 2. Otherwise, like Metadata Object 2, OSD retrieves the fingerprints from the chunk map, then it reads Chunk Object B by using the fingerprint (Fp B) as an OID. Finally, the OSD relays the data of Chunk Object B to Client 3.

Note that Ceph utilizes the CRUSH algorithm once again to calculate the chunk object location instead of maintaining a separate fingerprint index store. This technique is called double hashing [29].

2.4 Snapshot in Ceph

Ceph handles object snapshots via explicit APIs such as creation and deletion. OSD creates a new snapshot when a user makes a change on an object, at which point the version of the modified object is increased while keeping existing OID.

3 Motivation

3.1 Deduplication on unique chunks

In the prior approach, OSD decides whether an object needs to be deduplicated or not depending on which state the object is in between hot and cold instead of its redundancy; if the object is cold, OSD forces it to be deduplicated without checking its

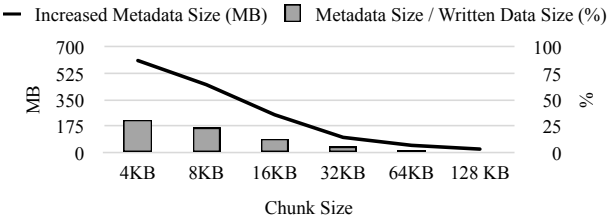


Figure 2: Metadata space overhead when deduplicating on unique objects. The total size of objects is 2 GB with 4 MB objects. Metadata size includes deduplication metadata (e.g., chunk information and fingerprint) and object (chunk and metadata) metadata.

redundancy. Therefore, the deduplication procedure inevitably generates unnecessary metadata even though the object has no redundant chunks. In the worst case, the amount of metadata could account for more than 30% of the total writes as shown in Figure 2. To eliminate the space overheads of the metadata, the best way is to remove only duplicate copies of chunks. To do so, however, we face the following two challenges:

Lack of knowledge of duplicate chunks. Ceph does not keep track of redundant objects. Moreover, to learn such information, each OSD would have to periodically check redundancy of all the objects in other OSDs, resulting in high overheads, interfering with foreground I/Os.

Object state and chunk-level management. The prior object management approach is unable to handle the cold object that has only unique chunks; when the object becomes cold, OSD deduplicates the cold object implicitly, degrading deduplication efficiency. To avoid this problem and maintain compatibility with the existing Ceph implementation, an additional state is needed to distinguish between unique and duplicate during the state transition from hot to cold. Moreover, there is a chance that an object has too few duplicate chunks to benefit from the reduced footprint of metadata. In this case, it's better not to deduplicate the object because the majority of the chunks are not redundant.

3.2 Structural limitations

Coarse-grained tiering mechanism. Ceph uses a coarse-grained tiering approach for deduplication; a background thread in each OSD updates the object state (hot or cold) and performs deduplication. However, it places a negative impact on the OSD foreground performance. In fact, to ensure consistency, Ceph holds a lock on PG even within an iterative loop for object enumeration to protect live objects in PG. Note that this PG lock is required when OSD handles a write operation to append a PG-level log for recovery. This is not a problem when the number of objects is small. However, if OSD contains a large number of objects, it is non-negligible because even simple object enumeration is very costly.

Moreover, the proposed tiering method does not expose the interfaces that allow other external modules to gather chunk information and trigger deduplication on demand. This strategy limits the effectiveness of deduplication.

No support for CDC. As an initial step, deduplication in Ceph was proposed based on FSC. FSC has the benefit of the low overhead calculation to determine chunk boundaries because it uses a predefined chunk size. However, it is well known that there is a boundary-shift problem [26] in some workloads. To avoid the problem, many studies have exploited CDC [10, 23, 24, 32]. We have tried complementing the fixed chunking's limitation using CDC. Unfortunately, applying CDC to the existing system leads to another challenge.

In the FSC approach, the user should configure the size of the chunk manually, then a background thread in OSD deduplicates the content of the chunk. This method is quite straightforward—there are only three operations: (1) set a chunk range (e.g., 8 KB chunk), (2) generate a fingerprint from the chunk, and 3) perform the content migration from the base tier to the chunk tier. In other words, once a range is set, the range remains valid until the range setting is changed. However, it does not seem well suited in terms of CDC because CDC generates different chunk sizes depending on the content; CDC may end up triggering multiple range changes even when a single chunk is mutated, so that the chunk ranges need to be recalculated.

Also, the prior approach uses three chunk states for deduplication: MISSING, DIRTY, and CLEAN. MISSING means that the content of the chunks does not exist in the base tier (not cached). DIRTY represents that the object was updated after the chunk was deduplicated, while CLEAN indicates that nothing has changed since the last deduplication operation. However, in CDC, there is no DIRTY state. If the chunk becomes dirty, its state becomes invalid as chunk range recalculation is required.

3.3 Inefficient reference management

The prior approach uses a false-positive based reference counting design [29]—while a chunk object is allowed to have a reference of deleted metadata object, the metadata object is guaranteed to always point to a valid chunk object—to prevent the failures. Nevertheless, it has other limitations, as follows:

Limitation of using the reference count. Reference mismatches may occur if OSD crashes and fails to successfully complete operations to update reference counts. To check reference mismatches in Ceph, the *scrub* process selects a chunk object, then scans all metadata objects to examine how many metadata objects have the reference of the chunk object; the expected reference count should be less than or equal to the reference count the chunk object holds. Note that the chunk object holds its own reference count, as explained in Section 2.3. Moreover, this behavior is repeated for all chunk objects, which in turn causes a scalability problem as the number of objects grows. The time complexity of the *scrub* process is $O(\#\text{chunk objects} \times \#\text{metadata objects})$.

Snapshot compatibility. In the previous approach, most of the features work with deduplicated metadata objects, but the snapshot feature does not, because reference management is missing for snapshot.

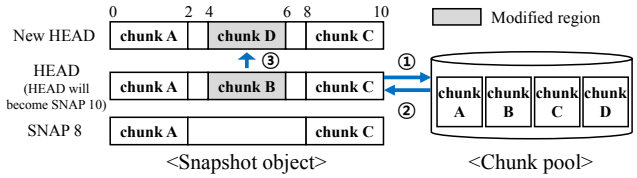


Figure 3: Ceph reference management (1. Send reference increment message, 2. ACK, 3. Update chunk information). SNAP represents a snapshot.

T	1	32	64	128	256	512
Latency (sec.)	0.13	0.32	0.72	1.8	3.96	9.37

Table 1: Snapshot creation time (T: the number of chunks on 4MB object, ten objects are used)

One straightforward yet naïve approach to supporting snapshot would be to increase (or decrease) the reference counts of all chunks of a deduplicated object whenever its snapshot is created (or deleted). However, this would generate an excessive number of messages among OSDs. Figure 3 describes the overhead. There is a HEAD that is the latest object version. For an incoming write to update the HEAD, OSD creates a snapshot (SNAP 10) in ascending order, excluding the new write. At this time, all chunk information within unmodified regions (chunks A and C) in the HEAD should be copied to the new HEAD's metadata ❸, after the reference increment (❶ and ❷) is done. Therefore, the snapshot creation will be delayed until OSD receives all ACKs for the reference increment. Note that an operation to increase reference count is synchronously done due to false-positive characteristics. Table 1 shows the snapshot creation time depending on the number of chunks. If there are 512 chunks on objects, the latency dramatically increases up to 9.37 seconds.

Aside from the delay, there is a redundant increase in the chunk's reference count if every snapshot has its own reference. For example, the reference count of chunks A and C will be three because new HEAD, SNAP 8 and SNAP 10 have the same chunks A and C in Figure 3. What is important is that even if the new HEAD, SNAP 8 and SNAP 10 are deleted, the reference count may not decrease due to the false-positive based reference counting strategy used. Although this reference leak can be fixed via the *scrub* process, it consumes additional space, until the *scrub* process is complete.

4 Design and Implementation of TiDedup

Figure 4 describes the overall structure of *TiDedup*. Basically, clients are not aware of the existence of the chunk tier and issue I/Os to the metadata objects in the same way as normal objects. For instance, the OSD stores content to the metadata object (W1) on a write request. In addition, upon a read request, OSD either returns the metadata object to the client immediately if it is cached (R1) or reads the content

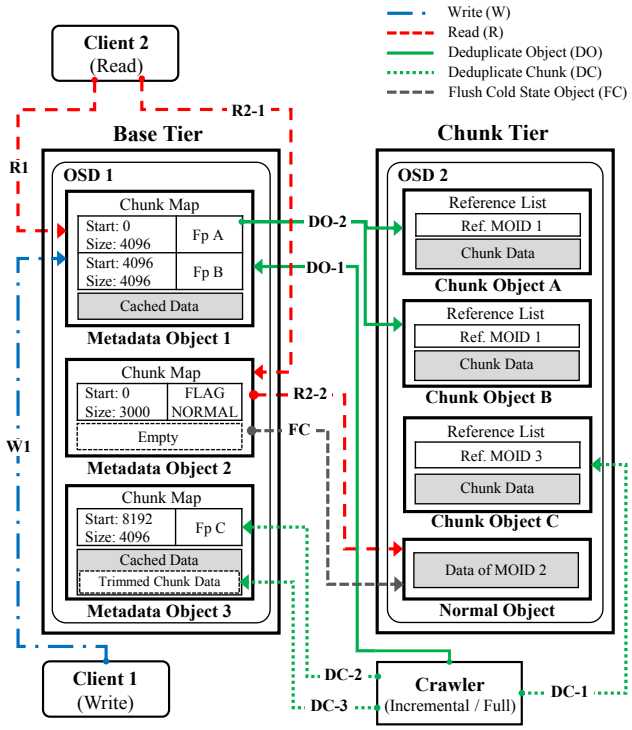


Figure 4: *TiDedup* architecture.

from the chunk object (R2-1 and R2-2). The crawler, which is responsible for triggering deduplication, scans objects—the scanning range is evenly divided and distributed to multiple threads in the crawler—on the base tier in either an incremental or full manner to look for redundant chunks. Then, the crawler checks the state of the metadata object using *stat*; *stat* retrieves the metadata object's information about hotness, deduplicated, and dirty. Depending on the state of the metadata object, the crawler works as follows.

- If the metadata object is hot, the crawler skips performing deduplication.
- If the metadata object is cold and contains more duplicate chunks than the threshold, the crawler performs deduplication if the object was not deduplicated before.
- If the metadata object is cold and has a few duplicate chunks, the crawler makes sure that the cached data is moved to the chunk tier (FC).

The crawler performs deduplication on the target object by using either *tier_flush* (DO1 and DO2 in Figure 4)—triggering CDC and moving chunks to chunk objects—or *set_chunk*—copying a target chunk to a chunk object in chunk tier (DC1) and setting a reference between metadata object and chunk object (DC2). Then, the crawler calls *tier_evict* to trim a range of the chunk in the metadata object, if necessary (DC3). These APIs will be described fully in Section 4.2.1.

4.1 Selective cluster-level crawling

To reduce the overhead as described in Section 3.1, we propose Selective cluster-level crawling, which crawls and carefully deduplicates objects on a base tier. The crawler is designed to run as a stand-alone application by decoupling a controller scheme—finding live objects and triggering deduplication—from OSD. Furthermore, we adopt incremental and full modes of crawling; both modes are introduced to figure out dedup-able objects on the base tier, but their crawling costs are different. In incremental mode, by choosing a small number of metadata objects gradually, the crawler reduces the chances of overutilizing system resources; during the daytime, it runs not to disturb user I/Os and unexpected high-priority I/Os, such as recovery and consistency checks. However, the incremental mode can not determine all dedup-able objects at once due to the limited range of search. To complement this weak point, full mode applies the full search without idle time. The drawback of the scheme is that it consumes more resources than the incremental mode does. Therefore, the crawler in full mode is scheduled to run once a week or at a longer interval, mostly during the nighttime when the system anticipates lower user I/O traffic.

In incremental mode, at first, the crawler gets a list of live objects sorted by OID on the base tier, then chooses a small group of objects from the list. Next, the crawler looks for redundant chunks among the selected objects—the crawler reads the objects, then runs CDC on the objects to calculate the fingerprint from the chunk, checking if the fingerprint is identical to other fingerprints collected before. The crawler considers the chunk a duplicate chunk if the chunk meets the condition; we define `chunk duplicate count`, which is the number of redundant chunks among the objects. If `chunk duplicate count` is higher than the threshold value K, the crawler regards the chunk as a good candidate for deduplication. Once duplicate chunks are found, the crawler has two tasks to do as follows. First, the crawler performs chunk-level deduplication (DC1, DC2, and DC3 in Figure 4). Second, the crawler checks information about how many duplicate chunks there are in an object. We call it `intra-object deduplication ratio`. The object, which has higher `intra-object deduplication ratio` than the threshold value L, is deduplicated via *tier_flush*, which is explained in the next section, by the crawler. The crawler pauses incremental mode when the tasks for the small group of objects are done, and resumes incremental mode, choosing the next small group of objects after the user-configured time T, 30 seconds by default. If there are no remaining objects in the list, the crawler refreshes a list of live objects. Then, it repeats the procedure in reversed direction while clearing the collected fingerprint list. In full mode, the differences from the incremental mode are the time interval and the search scope—it scans all objects in the base tier with a large time interval.

The crawler manages an in-memory fingerprint store to keep track of duplicate chunks; a key-value pair is used: ⟨fingerprint : redundant count⟩. The crawler updates the pair value when a fingerprint is calculated in either incremental

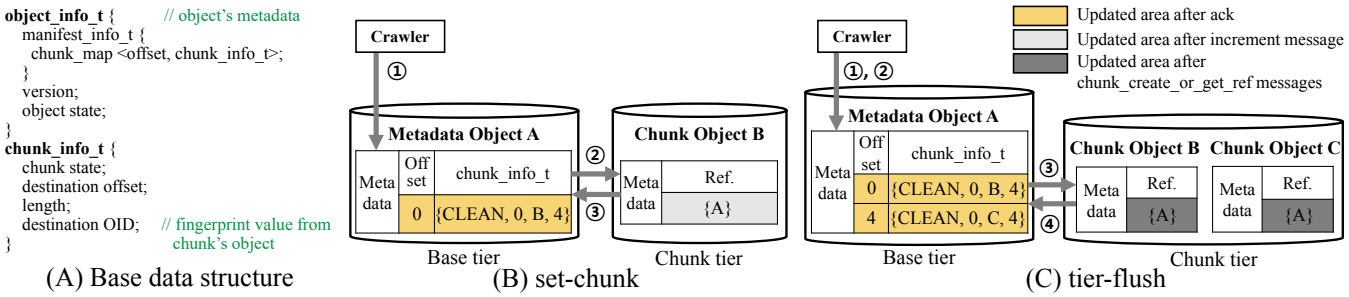


Figure 5: Base data structure and two API procedures (*set_chunk* and *tier_flush*).

or full mode. If memory usage exceeds a given threshold value configured at startup, the crawler deletes all entries in the fingerprint store, except for duplicate fingerprints—the number of redundancies is higher than the threshold—information. This strategy might lose deduplication opportunities; the crawler is likely to drop fingerprints that appear once in a while under memory pressure. However, the crawler can become stateless, which overcomes practical issues via simple re-execution.

Object Management. A metadata object in a base tier can be in one of the following states: hot, cold, or deduped. When an object is newly created, its initial state is set to hot by default; the hot object is not deduplicated because it is more likely to be mutated in the near future. The transition from hot to cold takes place over time when *TiDedup* determines that a certain hot object needs to be evicted based on LRU. Then, the cold object is deduplicated if the corresponding object contains more duplicate chunks than `intra-object deduplication ratio` by the crawler. Otherwise, the cold object is migrated to the chunk tier without deduplication process (no chunking and fingerprinting).

TiDedup promotes chunk objects to the base tier if they are either updated or frequently accessed by read operations to avoid decoding overhead of deduplication; the object state is changed to a hot state at this time. Since we decide not to use the background thread at all, as explained in the following section, OSD keeps the object state using existing in-memory object metadata (*object_info_t*) and updates its state when the object is accessed.

4.2 Event-driven tiering mechanism with CDC

We design event-driven tiering to process multiple requests concurrently and minimize interference to foreground I/Os. Event-driven tiering does not perform any background works that can affect incoming I/O requests. Instead, it exposes APIs to the crawler to perform deduplication on demand. In addition, we redesign the overall I/O path to support CDC.

Basic read and write. Upon a read request, OSD looks up the metadata of the corresponding metadata object (*object_info_t*) where *manifest_info_t*—metadata related to deduplication—is stored. As shown in Figure 5 (A), *manifest_info_t* has *chunk_map*, which is a map data structure including source offset and *chunk_info_t*, to maintain mapping information between the source and destination chunk. Plus, each *chunk_info_t* has a state variable that indicates either MISSING or CLEAN. If the chunk state is MISSING, OSD calls *tier_promote* to move the chunk object from the chunk tier to the base tier in advance of responding to the user. If the chunk state is CLEAN, OSD replies to the user with the existing content the metadata object has.

For a write request, before storing content to the metadata object, OSD clears the chunk information (*chunk_info_t* in *chunk_map*) within modified range, while sending delete messages; if the write request overwrites a whole range of the object, no *chunk_info_t* exists. This is because modifying content means that the corresponding chunks are no longer meaningful—all chunk boundaries should be recalculated by CDC.

4.2.1 APIs with CDC

Set_chunk. The purpose of *set_chunk* is to set a chunk boundary within a metadata object for deduplication to support a case that only a specific range of the metadata object is redundant. To do so, *set_chunk* stores the given input argument—<source OID, destination OID, source version, source offset, length, and destination offset>—to the corresponding *chunk_info_t* in *chunk_map*. *Set_chunk* has two main roles: (1) to increase the target chunk' reference count, and (2) to update the chunk information.

As soon as OSD receives a *set_chunk* message from the crawler (❶ in Figure 5 (B)), it sends a message to add the reference of the source object ❷. Since calling *set_chunk* implies that the source object takes a reference of the target chunk object, reference increment should be conducted before the chunk information is stored. We will describe reference management in more detail in the next section. After the reference increment is completed ❸, *TiDedup* updates *chunk_info_t*. For example, when a user calls *set_chunk* with <source OID, fingerprint OID, v2, 4096, 8192, and 4096>, the OSD adds *chunk_info_t* (CLEAN, 4096, fingerprint OID, 8192) to *chunk_map* at offset 4096 if the object's version is v2; note that OID includes tier information where the object is located and OSD increases the object's version number whenever the object is changed. The crawler is required to maintain the target version to deduplicate objects correctly.

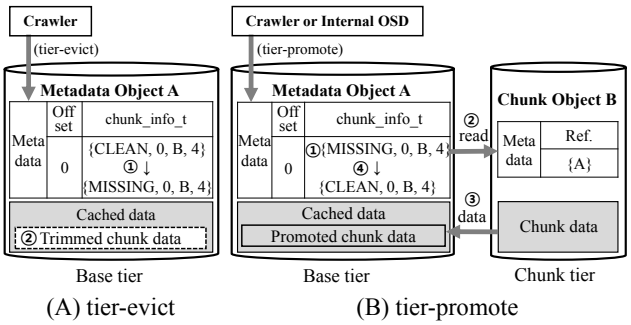


Figure 6: Procedure of *tier_evict* and *tier_promote*.

Set_chunk presumes that the target chunk is already copied to the destination OID before calling *set_chunk*, so the initial state of *chunk_info_t* is CLEAN.

Tier_flush. The crawler invokes *tier_flush* to remove redundancy on a metadata object, not a chunk; *tier_flush* deduplicates all contents on a metadata object explicitly if the object is not hot. Once *tier_flush* is called (❶ in Figure 5 (C)), OSD reads all contents from the metadata object, then executes CDC to generate chunks. After that, OSD generates a fingerprint value from each chunk ❷. Using given the fingerprint values as OID (B and C), OSD sends other OSDs in the chunk tier a compound operation, called *chunk_create_or_get_ref* ❸—this operation either increases the reference count if the target object is present or creates a new object with setting the reference count to one through transaction if the object does not exist. Once OSD receives all completion responses of the *chunk_create_or_get_ref*, it stores the metadata changes (e.g., *chunk_info_t*), and set the metadata object to deduped ❹. Although a failure can occur on rare occasions during *tier_flush*, *TiDedup* maintains consistency, because the metadata changes caused by *tier_flush* are not persistent until all the *chunk_create_or_get_ref* operations are completed. Note that *tier_flush* only updates metadata in terms of the metadata object. For example, if two out of ten *chunk_create_or_get_ref* are only successfully completed, *TiDedup* never updates the corresponding *chunk_info_t*. Instead, the crawler will retry to deduplicate the metadata object because it is not marked as deduped.

Tier_evict. *Tier_evict* removes the object's content using the punch-hole technique [20,22]. Figure 6 (A) demonstrates how *tier_evict* works. There is chunk B on the metadata object A. Once *tier_evict* is called, chunk B is marked as MISSING ❶, and then chunk B is trimmed from the metadata object A ❷, thereby resulting in a transition from a normal file to a sparse file. Note that the remaining parts of the metadata object A, which are not used as a chunk, remain on the metadata object A without removal. If a user tries to access the trimmed chunk, OSD in the base tier retrieves the original content from the chunk tier, before handling the user request.

Tier_promote. *Tier_promote* performs chunk migration from the chunk tier to the base tier even if a single chunk of metadata object is MISSING. Upon *tier_promote*, *TiDedup* finds chunks that have MISSING state in the metadata object ❶, as shown in Figure 6 (B), then sends read requests to corresponding chunk objects ❷. After the read requests are completed, *TiDedup* stores the given chunks in the base tier ❸. Unlike *tier_flush*, *tier_promote* updates chunk's content, and changes the chunk state from MISSING to CLEAN as soon as each read is succeeded ❹. For instance, although two out of ten reads are completed, *TiDedup* keeps the two completed chunks up-to-date.

4.3 OID shared reference management

To minimize *scrub* overhead and provide snapshot compatibility, we propose OID shared reference management design based on false-positive reference counting.

Data format for reference management. As described in the previous section, reference management using the number of references causes the *scrub* process to take a significant time to complete. To reduce the execution time for the *scrub* process, *TiDedup* makes use of Ceph's OID format to represent the reference instead of using a simple number. Since Ceph's OID contains location information, as described in Section 2.1, *TiDedup* can efficiently retrieve objects by their OID. In other words, if OID is used as a reference, *TiDedup* is able to recognize which metadata object refers to chunk objects like a back pointer and vice versa.

Scrub worker. *TiDedup* deploys a *scrub* worker as a separate thread in the crawler. *Scrub* worker wakes up periodically (the default value of wake-up period is equal to that in the full mode), then it begins to get a list of stored chunk objects on chunk tier and read their extended attributes one by one, each of which storing the reference information. Since the chunk object has its source OIDs, *scrub* worker does not need to read all metadata objects on the base tier to find reference mismatch. Instead, it just reads the metadata object whose OID is the source OID of the chunk object, then examines that the object has a reference to the chunk object. If the metadata object has a corresponding chunk object's OID, *scrub* worker repeats this until no more source OIDs that have not been checked present in the chunk object. If not, *scrub* process corrects the metadata object by removing the destination OID which is identical to the chunk object's OID.

Snapshot. To overcome snapshot-related limitations as described in Section 3.3, we introduce OID shared reference within an object. The key idea is that *TiDedup* does not generate an add/delete reference message if a chunk is identical—same offset, length, and destination OID—to a chunk in the adjacent snapshot. For instance, the reference count of BBB in Figure 7 is one; this means that only one add reference message is sent to chunk object BBB, because the chunk BBB in SNAP 7 is identical to chunk BBB in the new HEAD and SNAP 5, respectively. Note that we use the number of references here for the explanation.

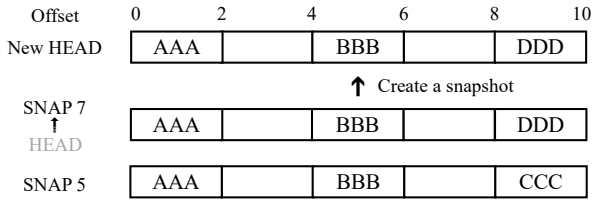


Figure 7: Snapshot reference management on snap creation.

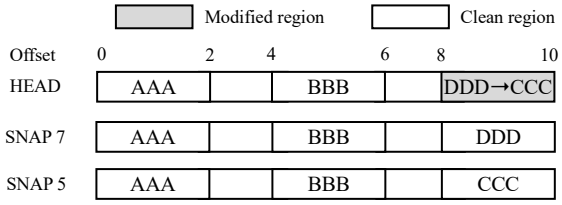


Figure 8: Snapshot reference management on overwrite (chunk DDD at offset 8 is changed to chunk CCC).

Figure 8 shows another example of a write operation. There is a modified region and clean regions in HEAD. *TiDedup* performs an add reference operation for chunk CCC—sending an add reference message then updating the corresponding *chunk_info_t* after the add reference message is done—, but does not generate a delete reference message for chunk DDD because SNAP 7 includes chunk DDD at offset 8. In addition, *TiDedup* does not do anything regarding all chunks in the clean region because each of the chunks is the same as the one in the previous snapshot.

However, a simple OID shared reference, as mentioned above, is insufficient to maintain consistency in reference management when the snapshot is removed. In Figure 8, SNAP 7 has a different chunk DDD compared to both SNAP 5 and HEAD at offset 8, so the reference count of chunk CCC is two, not one. At this point, if SNAP 7 is then removed, the reference count of chunk CCC will still remain two, even though it should be adjusted to one. To prevent this inconsistency, *TiDedup* checks both the prior snapshot and the next snapshot (SNAP 5 and HEAD in Figure 8) when the deletion occurs. If both chunks are identical, *TiDedup* sends a delete reference message.

As such, with OID shared reference, *TiDedup* generates only a limited number of add reference messages, regardless of many snapshot creations. Although *TiDedup* requires an additional search operation to identify the same chunks on the adjacent snapshots, this operation is relatively cheaper than handling the add reference operation.

OID shared reference can also work with proposed APIs, such as *set_chunk* (described in Section 4.2.1). In the *set_chunk* case; *TiDedup* exploits *set_chunk* to make a deduplicated chunk at any position in snapshots, as shown in Figure 9 (A), *TiDedup* performs an add operation for chunk AAA because there is no same chunk on the two snapshots (HEAD and SNAP 30), but nothing occurs in Figure 9 (B) because the HEAD includes chunk CCC. On the other hand, *TiDedup* generates a delete reference message for chunk CCC in Figure 9 (C) because both HEAD and SNAP 30 have the same chunk CCC.

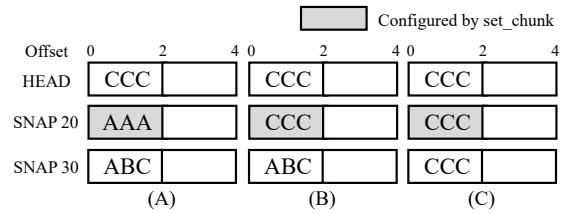


Figure 9: set_chunk (gray are new chunks by set_chunk).

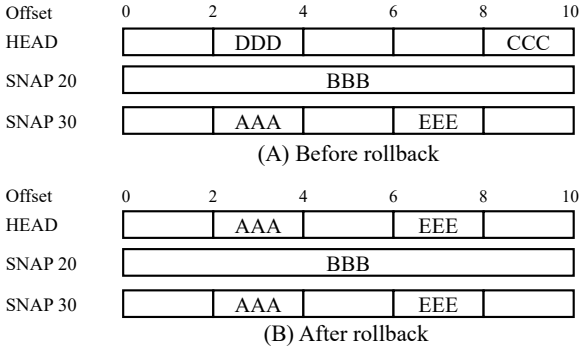


Figure 10: Rollback (HEAD is rollbacked to SNAP 30).

Rollback. *Rollback* replaces the HEAD with a given snapshot version. Upon *rollback*, *TiDedup* promotes all MISSING chunks both the HEAD and a given snapshot have from the chunk tier to the base tier. Then, the current HEAD is removed to make a correct clone into the HEAD. Figure 10 shows how shared reference count works with *rollback*. SNAP 30's chunks (chunks AAA and EEE) are copied to the HEAD (in Figure 10 (A)). Then, add operations for both AAA and EEE are needed because SNAP 20 has no identical chunks compared to the updated HEAD.

Crash consistency. OID shared reference management can maintain consistency after the crash because it is based on false-positive design. *TiDedup* guarantees the chunk object must exist if its metadata object has a reference. For this, the add operation for deduplicated chunks and update operation for relevant metadata are performed atomically. Moreover, *TiDedup* does not generate a delete reference message unless all shared chunk references in the object are unreferenced. Although *TiDedup* deletes *chunk_info_t* on sending a delete reference message without checking the completion of a delete operation—removing the metadata OID from the chunk object's extent attribute, this potential mismatch (a chunk object has a reference to a metadata object, but not vice versa) can be fixed by the *scrub* worker.

Reference set/get APIs. To set/get the reference, *TiDedup* adds four APIs that can be called by external clients and internal OSDs. *chunk_create_or_get_ref* creates a chunk object if the chunk object is not present, then adds a reference, which is the metadata OID. There are *reference_get* and *reference_put* to add/delete the reference of the chunk object. *read_reference* returns the list of references either the meta-

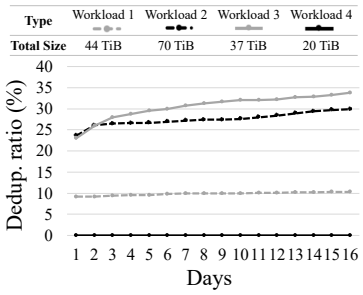


Figure 11: Space saving on factory data.

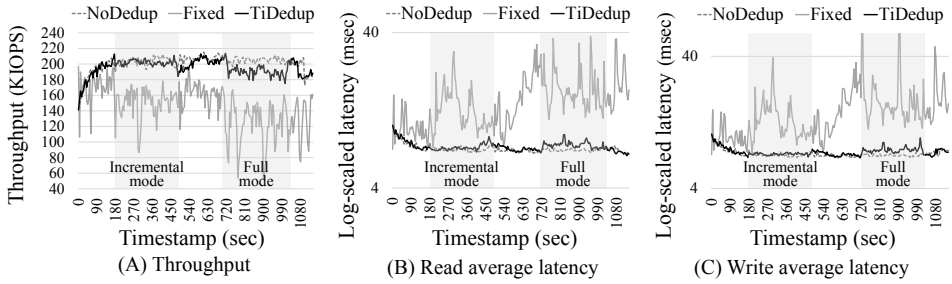


Figure 12: YCSB throughput (Workload a, record count is 500K, operation count is 20M).

data or the chunk object has. One important thing here is that the metadata object must return a reference list according to OID shared reference design; for example, in Figure 8, the reference list is {chunk AAA, BBB, CCC, CCC, and DDD}. On the other hand, the chunk object returns all references without considering OID shared reference.

5 Evaluation

5.1 Environmental setup

We use 10 machines in total, each of which is equipped with a 2-way AMD EPYC 7543 32-cores (80 threads) per NUMA node and 512 GB of DRAM. All nodes are connected with a 100 GbE network. Six nodes are used as storage nodes. Four nodes are clients to issue I/Os for evaluation. The latest version (Reef) of the Ceph storage cluster is configured by default parameters, except that the replication factor is two. Each machine has six BM1733 QLC SSDs (4 TB) and runs an OSD daemon on a single SSD. Note that we use high-performance equipments for evaluation to eliminate other performance factors. We use FastCDC [46] with the average chunk size set to 16KB. SHA1 is used to generate fingerprint value among the available fingerprint algorithm options (e.g., SHA1, SHA128, and SHA256). We set `Intra-object deduplication ratio` as 30% and `chunk duplication count` as four empirically. `NoDedup` represents default Ceph [42] without deduplication. `Fixed` means a prior deduplication approach for Ceph [29].

5.2 Deduplication ratio and throughput

5.2.1 Space saving

Internal cloud dataset. As shown in Table 2, `Fixed` shows limited data reduction compared to *TiDedup*. In `Logs` dataset, the data, which is partially mutated without data alignment, continues to be appended to Ceph cluster—the system monitor records similar logs from the cluster every 30 seconds, so that CDC is more effective than `Fixed`. In addition, CDC can save more space on `Virtual disks` dataset even if the dataset includes not only OS partitions but also user data. The reason is that the dataset is gathered from the internal developer cloud service, so user data includes duplicate data, such as mail, source code, and large images.

	Virtual disks			Logs		
Chunk size	8K	16K	32K	8K	16K	32K
Fixed	21%	12%	10%	5.7%	5.4%	5.3%
TiDedup	45%	36%	27%	18.5%	16%	12.6%

Table 2: Space saving on real-world datasets depends on the chunking algorithm and average chunk size. `Virtual disks` represents VMware vSphere images (10.1 TB) from a developer cloud service (67 users). `Logs` represents service logs (560 GB) for cloud infrastructure including monitoring and device state.

Factory dataset. We replay factory dataset generated during the semiconductor manufacturing on Ceph's object service (RGW). The factory dataset is normally used to detect or predict malfunctions of semiconductor products. We collect four types of data in total on a daily basis. As shown in Figure 11, *TiDedup* can achieve a high deduplication ratio (up to 30%) on Workload 2 (chip information during manufacturing). This is because Workload 2 has time-series monitoring logs having periodic values—the similar structured data is consistently appended. Unlike Workload 2, Workload 1 (equipment status) also includes time-series logs but has a smaller entry size—small tables with timestamps. This leads to less data reduction. Workload 3 (logs for photo lithography) has daily archive files stored in an incremental manner. It contains a large amount of redundant data and shows a high deduplication ratio. Workload 4 consists of metrology and inspection image files which are already compressed and contain little redundancy. Thus, it is not affected by deduplication at all.

5.2.2 Throughput

YCSB. We use YCSB workload a (read/write ratio is 50:50) to generate foreground I/Os. YCSB runs on four clients with sixteen threads using Ceph's block device service (RBD). As shown in Figure 12, the crawler is launched with incremental (at 180 seconds) and full mode (at 720 seconds), respectively—each of which runs for 300 seconds. Note that the elapsed time for incremental mode includes user idle time.

Figure 12 (A), (B), and (C) shows throughput and average latency. With *TiDedup*, throughput is not degraded significantly compared to `NoDedup` because *TiDedup* does not trigger the deduplication aggressively until the object is cold. *TiDedup* also achieves near-constant throughput unlike `Fixed`, which suffers a significant performance drop because a background thread blocks foreground OSD I/Os. Moreover, *TiD-*

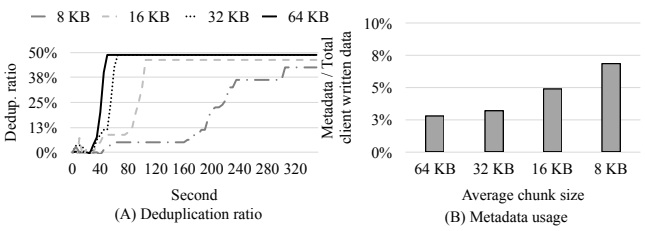


Figure 13: Space saving depending on the average chunk size.

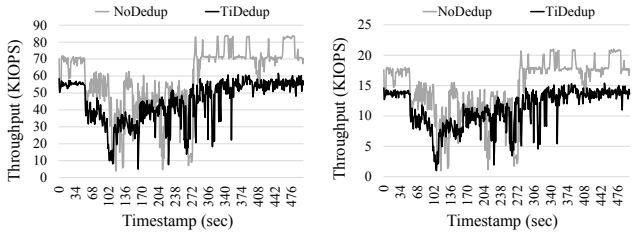


Figure 14: Recovery performance (Average IOPS of four clients).

edup with incremental mode has nearly no effect on both throughput and average latency due to a limited search scope. However, *TiDedup* with full mode badly affects performance, due to two reasons: (1) cold or deduped objects are accessed frequently in full mode, and, (2) YCSB's request can be blocked during *tier-flush*.

5.3 The impact of average chunk size

Figure 13 shows how much storage space can be saved depending on the average chunk size [46]. We generate 50% of redundant contents by `fio` with varying chunk sizes, then launch the crawler to perform deduplication. In the case of a large chunk (>16K), the amount of reduced data increases rapidly because the number of generated chunks is less than the small chunk's size, so the deduplication job can be done early. Also, we observe that the deduplication ratio does not reach 50% due to additional metadata for deduplication, as shown in Figure 13 (B). Interestingly, CDC generates non-aligned data, unlike FSC; for instance, a 8,200 byte chunk—not aligned by block size (4,096 byte)—can be generated by CDC, causing non-aligned data allocation. As a result, 12 KB is allocated even though the requested object size is 8,200 byte. This aggravates metadata consumption.

5.4 Worst-case recovery performance

We run fio with 8KB random read/write workload—the ratio between reads and writes is 8:2. During the mixed workload, the crawler issues bursty traffic—64 threads submit object reads and *tier_flush* concurrently—to the base tier, and triggers *scrub* using 16 threads. On top of that, 4 out of 36 OSDs are suddenly down during the test, resulting in generating recovery I/Os for data rebalance. Figure 14 (A) and (B) show *TiDedup*'s read and write IOPS over time. Overall, we observe performance fluctuation due to the bursty traffic from the crawler. At around 50 seconds, two OSDs are pulled out from the storage cluster and the performance decreases rapidly due to the recovery I/Os. After 50 seconds, two more OSDs are out and the fluctuation gets worse. However, at around 350 seconds, all four OSDs rejoin the cluster and the performance of foreground I/Os are recovered eventually. Compared to `NoDedup`, even in the worst case, *TiDedup* only reduces the performance by 25% on average. It is noteworthy that we can further mitigate foreground performance drop if the crawler employs a flow control technique in the event of congestion.

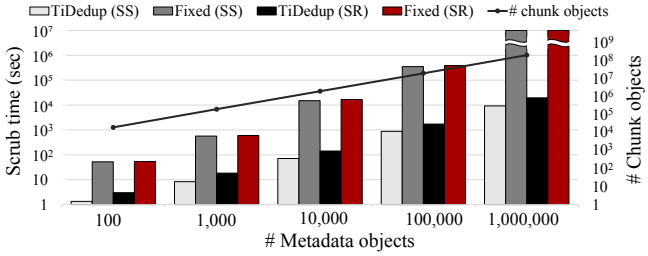


Figure 15: Scrub time comparison (SS: `Scrub-search`, SR: `Scrub-repair`). SR time includes SS time.

5.5 Scrub time

To compare *scrub* time between `Fixed` and *TiDedup*, we run a crawler, which spawns sixteen *scrub* threads (each thread issues six *scrub* requests asynchronously), with varying the number of metadata objects as shown in Figure 15. Each *Scrub* thread searches objects in the chunk tier and reads chunk information as described in Section 4. If the chunk object has a reference to the metadata object, the *scrub* thread reads either all metadata objects (`Fixed`) or a corresponding metadata object (*TiDedup*) in the base tier to check if the reference for chunk object is valid (`Scrub-search`). If the reference is invalid, *TiDedup* fixes the mismatch (`Scrub-repair`). In `Fixed`, *scrub* time increases considerably due to a full scan for metadata objects. *TiDedup* also takes considerable time for *scrub*, but the execution time is significantly reduced because *TiDedup* needs only a single read to check if the referenced metadata object is valid. In the case of one million metadata objects, *TiDedup* takes about four hours to complete, while we could not measure the `Fixed` *scrub* time because even after five days, the job had not been done.

5.6 Snapshot creation and deletion

Figure 16 (A) shows snapshot creation time (ten objects) as the number of deduplicated chunks grows. To measure snapshot creation time, we create deduplicated chunks on a metadata object and issue snapshot creation requests to the metadata object. In `Fixed`, the execution time to create a snapshot increases linearly because the more deduplicated chunks the snapshot has, the more operations (*chunk_create_or_get_ref*) OSD needs to handle. On the other hand, *TiDedup* takes a constant time even thanks to the OID shared reference count.

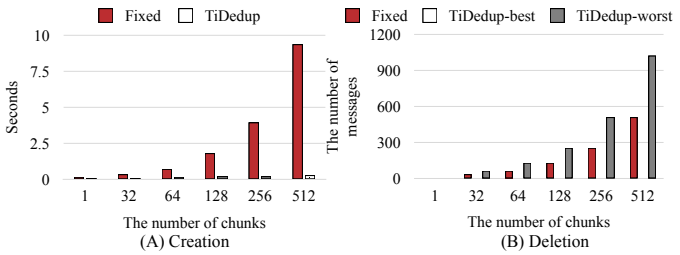


Figure 16: Snapshot creation/deletion comparison.

YAML	Description
dedup-io-mixed	read, write, set_chunk, tier_promote, tier_evict, tier_flush
dedup-io-snap	read, write, set_chunk, tier_promote, tier_evict, tier_flush, snap_create, snap_remove, rollback

Table 3: Integration test description.

Figure 16 (B) shows the number of generated messages from an OSD during snapshot deletion. *TiDedup*-best is the best-case scenario where no chunk differs from adjacent snapshots (e.g., chunk AAA at offset 0 in Figure 8). On the other hand, *TiDedup*-worst represents the extreme case where all chunks are different from adjacent snapshots and both the prior and next snapshots are identical (e.g., chunk DDD at offset 8 in Figure 8). In *TiDedup*-best, no messages are generated. However, compared to `Fixed`, *TiDedup*-worst generates a higher number of messages because *TiDedup* needs to reduce the references due to the same adjacent snapshots. But, these additional messages are proactively generated. Unlike *TiDedup*, `Fixed` would eventually generate the messages if further deletion occurs on either the prior or the next snapshot.

5.7 Integration test

We improve existing stress test coverage in Ceph to make *TiDedup* stable. With yaml as shown in Table 3 and the test code we added, Ceph's integration test framework, called *teuthology* [38], can perform the tests with/-without other tests (e.g., network disconnection and OSD fail) for reliability. Moreover, to ensure reference reliability, the workload generator also checks if the live chunk object's source reference is valid after all operations are done. *TiDedup* passes the combination tests conducted by *teuthology*.

6 Discussions

Small chunk vs. large chunk. Small chunk sizes (< 8KB) would result in numerous small chunk objects and potentially large space overheads. On the other hand, the use of a large chunk size may lead to a lower deduplication ratio. In order to hit a desirable trade-off point, *TiDedup* provides an *estimate* CLI, which shows how much storage space can be saved depending on options, to users. *TiDedup* allows the users to select suitable options (e.g., FSC, CDC, and chunk size) or even decide not to use deduplication at all.

How scalable is crawler? We did not fully address how our crawler design works at scale. In Section 5, we deploy a single application that has multiple threads for crawling. However, if the cluster has a large number of objects, this application may not be able to cover all objects properly. To solve this problem, the system administrator can deploy many crawlers on demand where each crawler is in charge of a sub-dataset of the storage cluster depending on the workload. This task is easily done by using container-based deployment because the crawler is stateless.

Additional space overhead for OID reference. Since the OID-shared reference scheme uses OID to represent a reference, chunk objects should maintain all metadata object's OIDs referring to the chunk objects, requiring more storage space than the reference count method. We limit the number of chunk object's references to a threshold value. *TiDedup* forces OSD to stop performing deduplication on the chunk objects in case their number of references is over the threshold value.

Applicability of *TiDedup*. *TiDedup* relies on two schemes that are prerequisites for integration with other distributed storage systems: (1) two separate address spaces (base and chunk tier), and (2) chunk object lookup by using a hash-based mapping between those two tiers. While the two address spaces scheme is easily applicable to other storage systems, the lookup method is tightly coupled with the hash-based object placement scheme [15, 19, 36, 43].

7 Lessons Learned

Rethinking a tiering mechanism in a distributed storage system with strong consistency. Ceph originally implemented the tiering structure where a background thread reads and migrates objects between tiers for cache tiering, so we anticipated that adding a deduplication feature on top of the existing architecture would be straightforward. However, we needed to consider recovery scenarios against a variety of failure types. Also, the deduplication jobs not only required holding a PG lock for an extended period, especially as the number of objects grew, but also introduced a new lock domain. Note that a strong consistency storage system, such as Ceph, handles I/O operations in strict order while holding locks. Unfortunately, this approach eventually led to an imbalance in I/O loads across different PGs, resulting in significant degradation of user-perceived performance and tail latency. Considering these challenges, we made the decision to deprecate the existing tiering mechanism. Instead, we opted to delegate the responsibility of the tiering mechanism within OSD to other components, such as the crawler.

Deduplication is promising only when data is dedup-able. In a distributed storage system, storing data entails duplicate copies to ensure availability through replication or erasure coding. However, this increases storage space overheads more than we expected due to the following two reasons: (1) additional metadata space for deduplication, and (2) unaligned existing metadata caused by the small size of the new meta-

Table 4: Comparison of previous deduplication architectures and this work.

	TiDedup	CephDedup [29]	Data Domain [9]	DupHunter [49]	Nitro [21]	idedup [35]
Processing	post	post	post	post	in-line	in-line
Selective dedup	O	X	X	O	X	O
Chunking	CDC, FSC	FSC	CDC	unknown	FSC	FSC
Scale	global	global	local	global	local	local
Interface	file/object/block		file	file	block	block
Storage type	general	general	backup	docker image	general	primary
Implementation	Ceph mainline	research only	proprietary	research only	research only	proprietary

data added for deduplication. Moreover, a general purpose storage system cannot predict in advance whether incoming data is always dedup-able or not. As a result, based on the observation, we decided to perform deduplication only if the storage system is able to secure enough free space after deduplication is done.

Flexible namespace architecture for *TiDedup*. A single global namespace (tier) is not optimal for efficiently handling different types of data streams, as each stream may have its own unique access pattern. Considering that a general-purpose storage system must cater to diverse workloads, we have designed a flexible namespace architecture that allows users to create custom namespaces tailored to their specific workloads. With this architecture, users are able to create one or more custom namespaces, enabling them to handle multiple data streams while maintaining isolation between them. For example, users can align their custom namespaces (base tiers) with the corresponding services, such as object, file, and block, while utilizing a shared chunk tier.

8 Related Work

Although there is rich literature on deduplication [1,2,5,8,13,27,40,41,48,50,51], few studies evaluate their architecture in terms of scalability in real distributed storage systems and/or open their code for third party reproduction of results. In the next, we touch on the most relevant studies to our work. Table 4 gives a quick summary of their key properties.

Data Domain Cloud Tier [9] proposes a deduplication architecture based on two tiers, where it performs deduplication in the local tier first, and then backs up data in the remote tier. Unlike our work, Data Domain Cloud Tier targets a local storage and does not selectively performs deduplication according to redundancy. *TiDedup* is a cluster-level solution with scalability as key design consideration. To that end, *TiDedup* reduces the fingerprint index lookup overheads by using double hashing [29] and pursues selective deduplication.

DupHunter [49] builds on a multi-layer tier and employs a cache algorithm utilizing domain-specific knowledge (container image registry). Also, it takes a selective deduplication approach. We note that DupHunter targets a specific environment where container images are stored and distributed by docker registry. On the other hand, *TiDedup* is designed for more general environments where file, object, and block services are needed. It is unclear how DupHunter addresses the fingerprint index problem [12,45] and whether the design can coexist with other existing features in a real storage system.

Deduplication could hurt read performance when an object is scattered throughout the cluster. Several studies suggest techniques (e.g., prefetching and rewriting) [3, 11] to overcome degraded read performance. Among them, we believe that caching is the most efficient way to resolve the read performance problem. *TiDedup* exploits a caching technique and migrates chunk objects from chunk tier to base tier if the chunk object is frequently accessed. As a result, hot data will be served quickly with no overhead whereas serving cold data entails forwarding overheads between base and chunk tier.

Inline deduplication [13,21,35] eliminates data redundancy immediately. However, *TiDedup* adopts a post-processing technique along with caching. By doing so, *TiDedup* employs on-demand deduplication, allowing for minimizing performance degradation. We believe that it is more suitable for a general purpose distributed storage system.

9 Conclusion

This paper presents *TiDedup* towards efficient cluster-level deduplication for a general-purpose distributed storage system. *TiDedup* incorporates three new design schemes to overcome the scalability issues found in a prior proposal. We have a complete, fully validated implementation of the design and have integrated *TiDedup* into the Ceph main branch. Our comprehensive evaluation study reveals that *TiDedup* achieves storage space savings without hurting the scalability of Ceph. We hope that our work will become a foundation on which further research and development are undertaken.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Vasily Tarasov, for their valuable comments. This work was supported in part by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2003618).

References

[1] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–9, 2009.

[2] Z. Cao, H. Wen, X. Ge, J. Ma, J. Diehl, and D. H. C. Du. TDDFS: A tier-aware data deduplication-based file system. *ACM Trans. Storage*, 15(1):4:1–4:26, 2019.

[3] Z. Cao, H. Wen, F. Wu, and D. H. C. Du. ALACC: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *USENIX Conference on File and Storage Technologies, (FAST)*, pages 309–324, 2018.

[4] CERN. CERN cloud home page. https://clouddocs.web.cern.ch/. Accessed 2023-06-09.

[5] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 77–90, 2011.

[6] Z. Chen and K. Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 291–299, 2016.

[7] CNCF. CNCF user survey 2020. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf. Accessed 2023-06-09.

[8] W. Dong, F. Douglis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *9th USENIX Conference on File and Storage Technologies (FAST)*, pages 15–29, 2011.

[9] A. Duggal, F. Jenkins, P. Shilane, R. Chinthekindi, R. Shah, and M. Kamat. Data domain cloud tier: Backup here, backup there, deduplicated everywhere! In *USENIX Annual Technical Conference (ATC)*, pages 647–660, 2019.

[10] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta. Primary data deduplication - large scale study and system design. In *USENIX Annual Technical Conference (ATC)*, pages 285–296, 2012.

[11] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (ATC)*, pages 181–192, 2014.

[12] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan. Design tradeoffs for data deduplication performance in backup workloads. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 331–344, 2015.

[13] Y. Fu, H. Jiang, and N. Xiao. A scalable inline cluster deduplication framework for big data protection. In *Middleware 2012*, pages 354–373, 2012.

[14] gke. google kubernetes engine. https://cloud.google.com/kubernetes-engine. Accessed 2023-06-09.

[15] glusterfs. glusterfs home page. https://www.gluster.org/. Accessed 2023-06-09.

[16] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *The Israeli Experimental Systems Conference (SYSTOR)*, page 7, 2009.

[17] S. Kaisler, F. Armour, J. A. Espinosa, and W. Money. Big data: Issues and challenges moving forward. In *Hawaii International Conference on System Sciences*, pages 995–1004, 2013.

[18] kubernetes. kubernetes home page. https://kubernetes.io/. Accessed 2023-06-09.

[19] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[20] E. Lee, Y. Han, S. Yang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. How to teach an old file system dog new object store tricks. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[21] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *USENIX Annual Technical Conference (ATC)*, pages 501–512, 2014.

[22] H. Liu, W. Ding, Y. Chen, W. Guo, S. Liu, T. Li, M. Zhang, J. Zhao, H. Zhu, and Z. Zhu. CFS: A distributed file system for large scale container platforms. In *International Conference on Management of Data*, page 1729–1742, 2019.

[23] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A study on data deduplication in hpc storage systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2012.

[24] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[25] J. Min, D. Yoon, and Y. Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[26] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 174—187, 2001.

[27] A. Nachman, G. Yadgar, and S. Sheinvald. GoSeed: Generating an optimal seeding plan for deduplicated storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 193–207, 2020.

[28] F. Ni and S. Jiang. Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems. In *ACM Symposium on Cloud Computing (SoCC)*, pages 220–232, 2019.

[29] M. Oh, S. Park, J. Yoon, S. Kim, K.-w. Lee, S. Weil, H. Y. Yeom, and M. Jung. Design of global data deduplication for a scale-out distributed storage system. In *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1063–1073, 2018.

[30] openstack. openstack home page. https://www.openstack.org/. Accessed 2023-06-09.

[31] openstack. openstack user survey. https://www.openstack.org/analytics/. Accessed 2023-06-09.

[32] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2004.

[33] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Conference on File and Storage Technologies (FAST)*, pages 89–101, 2002.

[34] D. R.-J. G.-J. Rydning, J. Reinsel, and J. Gantz. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 16, 2018.

[35] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *USENIX conference on File and Storage Technologies (FAST)*, page 24, 2012.

[36] Swift. Welcome to swift's documentation! https://docs.openstack.org/swift/latest/. Accessed 2023-06-09.

[37] TencentCloud. Tencent cloud home page. https://www.tencentcloud.com/solutions/tstack. Accessed 2023-06-09.

[38] Teuthology. Teuthology GitHub. https://github.com/ceph/teuthology. Accessed 2023-06-09.

[39] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *USENIX conference on File and Storage Technologies (FAST)*, page 4, 2012.

[40] C. Wang, Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue. NV-Dedup: high-performance inline deduplication for non-volatile memory. *IEEE Transactions on Computers*, 67(5):658–671, 2017.

[41] Q. Wang, J. Li, W. Xia, E. Kruus, B. Debnath, and P. P. Lee. Austere flash caching with deduplication and compression. In *USENIX Annual Technical Conference (ATC)*, pages 713–726, 2020.

[42] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2006.

[43] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE Conference on Supercomputing*, pages 31–31, 2006.

[44] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[45] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based Near-Exact deduplication scheme with low RAM overhead and high throughput. In *USENIX Annual Technical Conference (ATC)*, 2011.

[46] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *USENIX Annual Technical Conference (ATC)*, pages 101–114, 2016.

[47] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou. AE: an asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345, 2015.

[48] Y. Zhang, W. Xia, D. Feng, H. Jiang, Y. Hua, and Q. Wang. Finesse: Fine-Grained feature locality based fast resemblance detection for Post-Deduplication delta compression. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 121–128, 2019.

[49] N. Zhao, H. Albahar, S. Abraham, K. Chen, V. Tarasov, D. Skourtis, L. Rupprecht, A. Anwar, and A. R. Butt. DupHunter: Flexible high-performance deduplication for docker registries. In *USENIX Annual Technical Conference (ATC)*, pages 769–783, 2020.

[50] X. Zou, W. Xia, P. Shilane, H. Zhang, and X. Wang. Building a high-performance fine-grained deduplication framework for backup storage with high deduplication ratio. In *USENIX Annual Technical Conference (ATC)*, pages 19–36, 2022.

[51] X. Zou, J. Yuan, P. Shilane, W. Xia, H. Zhang, and X. Wang. The dilemma between deduplication and locality: Can both be achieved? In *USENIX Conference on File and Storage Technologies (FAST)*, pages 171–185, 2021.

A Artifact Appendix

Abstract

We provide the artifact that includes the source code and instructions to explain how to run *TiDedup* on Ceph. The artifact also includes a document to describe how *TiDedup* is applied to Ceph.

Scope

TiDedup is a cluster-level deduplication architecture for Ceph, so the goal of the artifact is to describe how to perform deduplication on Ceph cluster using either the artifact or mainline Ceph. As explained in README.md, we provide instructions to build, deploy, and run *TiDedup* on Ceph. In addition, the README.md also provides an explanation of how to run *TiDedup* using the latest Ceph without the artifact.

Contents

The artifact contains *TiDedup*'s source code integrated into Ceph and a README.md file to build source code and run Ceph while enabling deduplication.

Hosting

TiDedup is available at https://github.com/ssdohammer-sl/ceph/tree/tidedup with detailed instructions. Since *TiDedup* has been merged to Ceph mainline, the source code is also available at https://github.com/ceph/ceph.

LoopDelta: Embedding Locality-aware Opportunistic Delta Compression in Inline Deduplication for Highly Efficient Data Reduction

Yucheng Zhang[†*], *Hong Jiang*[‡], *Dan Feng*[*], *Nan Jiang*[§], *Taorong Qiu*[†], *Wei Huang*[†]
[†]*School of Mathematics and Computer Sciences, Nanchang University*
[‡]*Department of Computer Science and Engineering, University of Texas at Arlington*
[*]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology*
[§]*School of Information Engineering, East China Jiaotong University*
Corresponding author: dfeng@hust.edu.cn

Abstract

As a complement to data deduplication, delta compression further reduces the data volume by compressing non-duplicate data chunks relative to their similar chunks (base chunks). However, existing post-deduplication delta compression approaches for backup storage either suffer from the low similarity between many detected chunks or miss some potential similar chunks, or suffer from low (backup and restore) throughput due to extra I/Os for reading base chunks or add additional service-disruptive operations to backup systems.

In this paper, we propose LoopDelta to address the above-mentioned problems by an enhanced embedding delta compression scheme in deduplication in a non-intrusive way. The enhanced delta compression scheme combines four key techniques: (1) dual-locality-based similarity tracking to detect potential similar chunks by exploiting both logical and physical locality, (2) locality-aware prefetching to prefetch base chunks to avoid extra I/Os for reading base chunks on the write path, (3) cache-aware filter to avoid extra I/Os for base chunks on the read path, and (4) inversed delta compression to perform delta compression for data chunks that are otherwise forbidden to serve as base chunks by rewriting techniques designed to improve restore performance.

Experimental results indicate that LoopDelta increases the compression ratio by 1.24∼10.97 times on top of deduplication, without notably affecting the backup throughput, and it improves the restore performance by 1.2∼3.57 times.

1 Introduction

Data deduplication has been widely used in computer systems to improve storage space and network bandwidth efficiency [26, 29, 36, 53]. Typically, it removes duplicate data at the chunk granularity (e.g., 8KB size) but fails to eliminate redundancy among (highly) similar but non-duplicate data chunks. Delta compression has been employed to further remove redundant data from post-deduplication non-duplicate but similar chunks, by compressing non-duplicate data chunks relative to their similar chunks (base chunks) [18, 19, 21, 37, 47, 48, 55].

In this paper, we focus on adding delta compression to inline backup storage which usually adopts only deduplication for data reduction. Some efforts have been made to achieve this. Shilane et al. [38] suggested that delta compression achieves more than 2× additional compression on top of deduplication, but it incurs extra I/Os for reading base chunks from storage, which significantly reduce backup throughput. One solution to this problem is to replace the hard disk drive (HDD) with other media with higher random I/O performance, such as solid state drive (SSD). However, this method is not cost-effective because HDD remains significantly cheaper than SSD. So, this paper focuses on HDD-based backup systems.

Zou et al. [55] proposed MeGA to perform delta compression for data chunks whose base chunks can be detected in the last and the current backups on top of deduplication. MeGA is developed based on the assumption that, after each backup, the system will reorganize both the data chunks and the base chunks of the delta-compressed chunks of this backup into a delta-friendly data layout. However, this assumption has a nontrivial impact on both the users and the backup systems, i.e., the backup operations for users are suspended in the reorganization process, and the backup system must perform service-disruptive reorganizations frequently and completes them promptly after each user's backup. This paper focuses on adding delta compression seamlessly to deduplication systems in a non-intrusive and non-service-disruptive manner.

Existing schemes that add delta compression to deduplication systems face many challenges. The first challenge is a low compression ratio. The base chunks required for delta compression are detected by indexing the sketches of chunks, and the compression ratio mainly relies on the strategies adopted to index the sketches [6, 11, 24, 32, 37]. Usually, sketches are weak hashes of the chunk data and can be used to detect similar chunks [4, 6, 11, 21]. Full indexing (indexing the sketches of all data chunks in storage) is the simplest indexing strategy. Our study in Section 3.1 suggests that when similar chunks appear within the same backup, this technique may decrease the compression ratio due to the selection of suboptimal base chunks.

Besides full indexing, the existing sketch indexing techniques can be classified into two categories: the logical-locality-based indexing [42, 55] and the physical-locality-based indexing [37, 38], which have complementary capabilities of detecting similar chunks. The former can detect most of the highly similar chunks by leveraging the logical locality between two adjacent backups but may miss some potential similar chunks; the latter can detect most of the potentially similar chunks by exploiting the physical locality preserved in storage units called containers but suffers from the low similarity of detected chunks. Our analyses in Section 3.1 indicate that the desirable properties of the two techniques can be combined by exploiting both the logical and physical locality.

The second challenge of adding delta compression to deduplication systems is extra I/Os for reading base chunks during backup, which prevent delta compression from being used in high-performance inline backup systems [37, 38]. A typical deduplication-based backup system organizes data chunks into containers each of which consists of hundreds or thousands of data chunks as the storage units [17, 53]. For such a system, a routine operation during deduplication is to prefetch metadata from containers being deduplicated against to accelerate duplicate detection. Our study in Section 3.2 demonstrates that containers holding potential similar chunks detected by using both the logical and physical locality can be prefetched by piggybacking on the routine operations without extra I/Os.

The third challenge is extra I/Os for reading base chunks during restore, which reduce restore performance significantly. More specifically, when delta compression is applied, data chunks may refer to previously written deltas, and the base chunks of these deltas may require extra I/Os during restore. To address this issue, it is necessary to identify such previously written deltas and avoid referring to them. Our analyses in Section 3.2 suggest that existing approaches to identify such deltas either are vulnerable to garbage collection (GC) or require extra I/Os. Our analyses also suggest that the previously written deltas whose base chunks require extra I/Os during restore can be predicted during backup by using the metadata prefetched by the routine operations during deduplication.

The fourth challenge is the potential to miss base chunks when rewriting techniques are applied. Backup systems often adopt rewriting techniques to identify infrequently reused containers and give up deduplicating against them to alleviate chunk fragmentation [14, 16]. To cooperate with rewriting techniques, data chunks in infrequently reused containers cannot serve as base chunks for delta compression, resulting in a compression loss. Our analyses in Section 3.3 suggest that if the target of delta compression is changed to previously written chunks, rather than data chunks in the ongoing backup as in the traditional delta compression method, to generate additional encoded copies of the previously written chunks, the un-encoded copies of previously written chunks can be eliminated during GC to achieve data reduction, which is equivalent to the effect of delta compression.

With the above observations, we propose LoopDelta based on the deduplication strategy that groups data chunks into containers and prefetches metadata from them for duplicate detection during deduplication [17, 53]. By combining the following four key techniques, LoopDelta embeds delta compression in inline deduplication non-intrusively.

- **Dual-locality Similarity Tracking.** LoopDelta tracks data chunks and base chunks of delta-compressed chunks of the immediate predecessor backup to capture highly similar chunks by leveraging the logical locality and tracks the containers holding the aforementioned data chunks to capture similar chunks stored in these containers by exploiting the physical locality.
- **Locality-aware Prefetching.** LoopDelta prefetches base chunks by piggybacking on routine operations to prefetch metadata during deduplication, thereby avoiding extra I/Os for reading base chunks on the write path.
- **Cache-aware Filter.** LoopDelta identifies the previously written deltas whose base chunks require extra I/Os during restore with the assistance of the recently prefetched metadata during deduplication and avoids referring to such deltas, thereby eliminating extra I/Os for reading base chunks on the read path.
- **Inversed Delta Compression.** For data chunks whose detected similar chunks are forbidden to serve as base chunks by rewriting techniques, LoopDelta delta-encodes the detected similar chunks relative to these data chunks while deferring the removal of the data of these delta encoded chunks to the GC process.

Experimental results based on real-world datasets indicate that LoopDelta significantly increases both the compression ratio and restore performance on top of deduplication, without notably affecting backup throughput.

2 Background

2.1 Data Deduplication

Deduplication and Restore Processes. Typically, a backup stream is divided into data chunks, which are fingerprinted with a secure hash (e.g., SHA1) [12, 31, 33, 35, 46, 49]. Each fingerprint is queried in a fingerprint index to determine whether the system already stores a copy of the fingerprinted data chunk. If true, the system does not store the data chunk but refers it to the previously written copy. Meanwhile, consecutive unique data chunks are grouped into a large I/O unit, called a container, and written to HDDs.

When a backup completes, a *recipe* recording the fingerprint sequence of the backup stream is stored for the future restoration [16]. When a stored backup is requested, the restore process accesses data chunks one by one according to their order in the recipe to reconstruct the original data. In this

process, a *restore cache* is maintained in memory [7]. The read unit in this process is a container.

Redundancy Locality in Backup Workloads. Backup workloads typically consist of a series of copies of primary data [3,28,41,43]. *Redundancy locality* in the backup workloads refers to the repeating patterns of the redundant data among consecutive backups [16]. The repeating pattern before deduplication is called *logical locality*, which is preserved in the recipe and sequence of consecutive data chunks before deduplication. That repeating pattern after deduplication is called *physical locality* (also called *spatial locality* [53]), which is preserved in containers.

Both categories of the locality have been widely exploited to improve deduplication performance [9, 17, 23, 27, 44, 53]. For example, a fingerprint index mapping fingerprints to the physical locations of the chunks is required for detecting duplicates. However, storing the index in HDDs would result in low backup throughput, while putting it in memory would limit system scalability. Zhu et al. [53] put the index in HDDs and alleviate the indexing bottleneck by using physical-locality-based caching. Lillibridge et al. [23] and Guo et al. [17] put the index in memory and reduced its memory footprint by using logical-locality-based sampling and physical-locality-based sampling.

Chunk Fragmentation and Rewriting. Deduplication renders data chunks of a backup stream to be physically scattered, and this is known as *chunk fragmentation* [14, 22, 30, 56]. Chunk fragmentation decreases the locality and efficiency of the techniques exploiting this locality. For example, it decreases restore performance and backup throughput of container-based backup systems [2, 22]. Backup systems often adopt rewriting approaches to identify *infrequently reused containers* and give up deduplicating against them to alleviate fragmentation [7, 8, 15, 20, 22, 39]. Here, the infrequently reused containers are previously written containers that contain only a few data chunks referenced by the current backup. The data chunks that refer to previously written data chunks stored in infrequently reused containers are called *fragmented chunks*, which will be stored (rewritten) along with unique data chunks to improve the locality of the current backup.

Among rewriting approaches, Capping [22] processes the backup stream in non-overlapping segments, each of which contains a sequence of consecutive data chunks. Within a segment, data chunks can refer to at most T old (previously written) containers. If the number of old containers exceeds T, only the most referenced T containers can be referenced, and data chunks referring to other old containers are rewritten to reduce fragmentation. The value T, also called *capping level*, is a configurable parameter that trades deduplication for restore performance.

2.2 Post-deduplication Delta Compression

Post-deduplication delta compression consists of three stages: (1) resemblance detection (finding similar candidates), (2) reading the base chunks, and (3) delta encoding.

Resemblance Detection. For data chunks not removed by deduplication, a sketch calculation approach calculates sketches for data chunks [32, 51, 54]. Sketches are usually weak hashes of the chunk data [4, 6, 11, 21]. Two chunks are considered similar if they have the same sketches. The sketch indexing strategy has a critical impact on resemblance detection efficiency, which will be discussed in Section 3.1.

Reading the Base Chunks. Reading the base chunks for an HDD-based system is the performance bottleneck. Shilane et al. [37, 38] indicated that I/O overheads required to read back the base chunks decrease the backup throughput to an unacceptable level. MeGA [55] monitors the containers holding the base chunks and does not perform delta compression for the data chunks whose base chunks are stored in rarely referenced containers to reduce I/Os for reading base chunks. Besides, PFC-delta [52] prefetches base chunks by piggybacking on the routine I/Os during deduplication.

Delta Encoding. Xdelta [25] is a popular delta encoding technique that employs hashing and indexing to identify and eliminate repeated strings between the target and base chunks. Edelta [45] simplifies this process by replacing some of the hashing and indexing operations with fast byte-wise comparisons through exploiting fine-grained locality between similar chunks.

2.3 Garbage Collection

Garbage collection (GC) removes invalid chunks (chunks not referenced by any unexpired backups) from the system to consolidate free space [5, 14, 17, 40, 56]. Duplicate chunks will be removed in the GC process [2, 10]. To improve write performance, a deduplication-based backup system might choose to write (rewrite) occasional duplicate chunks while deferring deduplication to a GC process [2].

GC first traverses live backups and marks the live chunks. For a data chunk with multiple physical instances, GC marks one (often the most recently written one) of them as the live chunk [10]. Then, it copies live chunks from partially-invalid containers to form new containers. Then, previous containers whose live chunks are copied out of are reclaimed.

3 Observations and Motivations

3.1 Analysis of Sketch Indexing Efficiency

Besides the full indexing technique, existing sketch indexing techniques can be divided into two categories: logical-locality-based indexing techniques and physical-locality-based indexing techniques. This section analyzes the efficiency of these two categories of indexing techniques as well as the full indexing technique.

3.1.1 Logical-locality-based Sketch Indexing

Substantial similar chunks for delta-compressing a backup can be detected from data chunks of its immediate predecessor backup and similar chunks of this backup. This is because

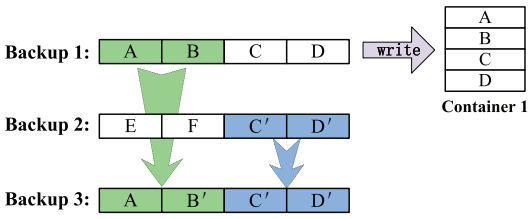


Figure 1: Chunks B', C', and D' are respectively similar to chunks B, C, and D. For backup 3, A and B' inherit from backup 1, while C' and D' inherit from backup 2. When detecting similar chunks for data chunks of backup 3, the logical-locality-based indexing techniques will miss B in backup 1, which is actually similar to B'.

each backup is often a modified copy of the last backup. Based on similar observations, MeGA [55] and HARD [42] index sketches of the data chunks and base chunks of the delta-compressed chunks of the last backup for resemblance detection. These sketch indexing techniques essentially exploit the logical locality between two adjacent backups.

An advantage of these indexing techniques is the high similarity of base chunks. In most cases, the best base chunk for the delta compression of a data chunk is its previous copy in the last backup because similar chunks are usually stemming from small edits to the last backup. A disadvantage of these indexing techniques is that they may miss some potential similar chunks. We observe that data chunks of a backup can inherit from multiple previous backup versions, for example, when data rollback occurs. This means that similar chunks may exist across backup versions. These similar chunks do not have a direct relationship to the immediate predecessor backup and thus cannot be detected by logical-locality-based indexing techniques. Figure 1 gives an example to illustrate how this problem may arise.

3.1.2 Physical-locality-based Sketch Indexing

Stream-Informed delta compression (SIDC) [38] is a physical-locality-based sketch technique that detects similar chunks by exploiting the physical locality among backups. Since physical locality is preserved in containers, SIDC is built on container-based deduplication systems. When a duplicate chunk is detected, the container holding the most recently written instance of this duplicate is selected for deduplicating against, and sketches of all data chunks in this container are indexed for matching similar chunks.

An advantage of this indexing technique is that it can detect most similar chunks including the ones missed by logical-locality-based indexing techniques. For data chunks that are inherited from versions older than the immediate predecessor backup, their sketches would be indexed if they are stored in the containers that would be deduplicated against, even if they do not have a direct relationship to the last backup. As shown in Figure 1, when the system deduplicates chunks in backup 3, container 1 is selected for deduplicating against because it holds the previously written copy of chunk A. Therefore, the

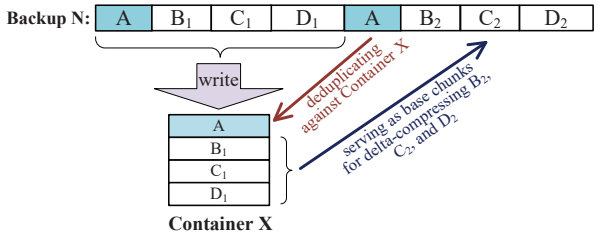


Figure 2: A in backup N is a self-referenced duplicate chunk, and the rest are self-referenced similar chunks. B_1, C_1, and D_1 are similar to B_2, C_2, and D_2, respectively. The first four data chunks are stored in container X. When deduplicating the second A, the system will select container X for deduplicating against because this container holds the previous-written copy of A. Then, sketches of data chunks in container X are indexed and B_1, C_1, and D_1 are detected as the base chunks for delta-compressing B_2, C_2, and D_2.

sketches of B in this container will be indexed and detected as a similar chunk of B'.

A backup may contain duplicates and similar chunks in itself, where the former are referred to as *self-referenced duplicate chunks* and the latter as *self-referenced similar chunks*. Compared with self-referenced similar chunks, similar chunks detected from the previous backups tend to share more redundancy with unique chunks of the on-going backup. This is because the latter are much more likely to result from the former after they are slightly modified. Consequently, the physical-locality-based sketch indexing technique can be suboptimal for datasets containing self-referenced duplicate and self-referenced similar chunks. This is because self-referenced duplicate chunks may cause the newly-written containers holding data chunks of the on-going backup to be selected for deduplicating against, which further causes the self-referenced similar chunks to be matched as the base chunks for delta compression. Figure 2 presents a simplified example to illustrate how the problem may arise.

3.1.3 Full Sketch Index

The full sketch indexing technique indexes sketches of all data chunks in the backup system and it is often serves as an upper bound for compression ratio evaluations when delta compression is involved [42, 52, 55]. Since the size of sketch indexes grows with the number of backup versions, it is challenging to organize the sketch indexes. Maintaining them in memory would limit system scalability, while putting them in HDDs would greatly reduce query throughput.

Another problem facing this technique is that it can be suboptimal for datasets containing self-referenced similar chunks. This technique indexes sketches of all data chunks in the system, including self-referenced similar chunks. Thus, when self-referenced similar chunks are ingested, the sketches of the best base chunk candidates for delta compression in previous backups may be replaced by those of self-referenced similar chunks, causing self-referenced similar chunks to be

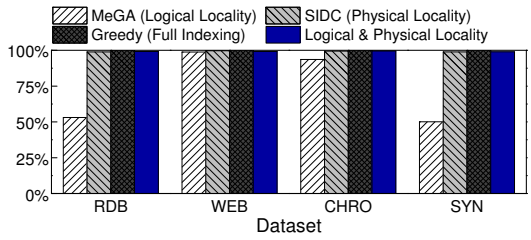


Figure 3: Percentage of potential similar chunks detected by MeGA, SIDC, Greedy, and the approach exploiting both logical and physical locality on four datasets.

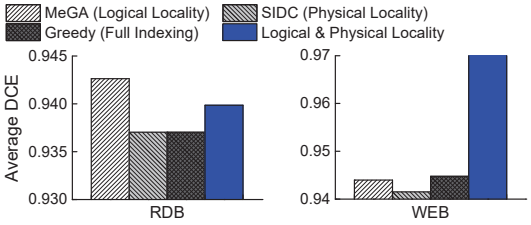


Figure 4: Average DCE of MeGA, SIDC, Greedy, and the approach exploiting both logical and physical locality on the RDB and WEB datasets.

matched as base chunks for delta compression.

3.1.4 Combining the Best of Both Worlds

Figures 3 and 4 respectively show the percentage and the average delta compression efficiency (*DCE*) [51,54] of detected similar chunks of existing sketch indexing techniques including MeGA, SIDC, and the full indexing (Greedy) for datasets whose characteristics are detailed in Table 1 in Section 5.1. Multi-version inheritance is common in RDB and SYN datasets, and the WEB dataset contains substantial self-referenced duplicates and similar chunks. *DCE*, which is calculated as $1 - \frac{chunk\ size\ after\ delta\ compression}{chunk\ size\ before\ delta\ compression}$, measures the similarity of the detected similar chunks. A larger value of *DCE* indicates higher similarity.

The results in the two figures agree with the earlier analysis in this section, i.e., the logical-locality-based technique can detect most highly similar chunks but may miss potential similar chunks, while the physical-locality-based technique can detect most potential similar chunks but suffers from low similarity. We found that the two categories of techniques have complementary capabilities for detecting similar chunks. This motivates us to propose a dual-locality-based sketch indexing approach to combine the advantages of both logical and physical locality. The dual-locality-based approach can not only detect most potentially similar chunks but also ensure high similarity between detected chunks, as shown in Figures 3 and 4. Since the full indexing technique can be suboptimal for datasets containing self-referenced similar chunks, the dual-locality-based approach can achieve higher *DCE* than the full indexing on such datasets.

3.2 Avoiding I/Os for Reading Base Chunks

Extra I/O overheads for reading base chunks on both the write and read paths prevent delta compression from being used in high-performance backup systems. In this subsection, we discuss and analyze the possible approaches to reducing or eliminating I/O overheads for reading base chunks to make delta compression feasible and practical for backup systems.

3.2.1 On the Write Path

For container-based deduplication systems, such as Data Domain backup systems [53], a routine operation during deduplication is to access containers for prefetching metadata to accelerate duplicate detection, which provides an opportunity to eliminate I/Os for reading base chunks. If the containers holding similar chunks will be accessed for prefetching metadata during deduplication, base chunks can be prefetched by piggybacking on the routine operations during deduplication without requiring extra I/Os.

Fortunately, if similar data chunks are detected by exploiting both logical and physical locality, as suggested in Section 3.1.4, due to redundancy locality, most of the containers holding potential similar data chunks would be prefetched during the metadata prefetching process in data deduplication. As a result, potential similar chunks can be prefetched, i.e., piggybacked on the retrieval of the metadata to serve as potential base chunks, thereby avoiding extra I/Os for reading base chunks on the write path. Zhang et al. [50] employed a similar base-chunk prefetching technique based on different observations, but their approach can only be applied to specific backup datasets (i.e., packed datasets containing substantial small files) and detects similar chunks only when rewriting is applied. In contrast, our approach can be applied to all backup datasets and can work without rewriting.

3.2.2 On the Read Path

During restore, base chunks of deltas also need to be read from storage for delta decoding. If base chunks are prefetched along with metadata during deduplication, they would be prefetched along with other chunks (or deltas) during restore, without requiring extra I/Os. However, when a data chunk refers to an old (previously written) delta, the base chunk of this delta may require extra I/Os during restore and decrease restore performance. Data chunks that refer to old deltas whose base chunks trigger read operations during restore are referred to as *base-fragmented chunks*, which should be rewritten for improved restore performance.

Existing approaches either are vulnerable to GC or require extra I/Os. SDC [52] can identify base-fragmented chunks by simulating the data restore process using container IDs during backup. However, it requires knowledge of the container IDs of the base chunks of referenced old deltas, which is difficult to obtain. Storing container IDs of base chunks along with deltas in containers is vulnerable to GC because base chunks can move around due to GC. Storing fingerprints of base chunks and obtaining their container IDs through the fingerprint index and cache may incur a large number of extra I/Os. Rewriting techniques, which are employed to identify fragmented chunks for deduplication systems, face the same

problem in identifying base-fragmented chunks because they also need the container IDs of base chunks for calculating the containers' reuse ratios.

When excluding base chunks, the order in which data chunks are processed during backup is the same as that during restore. Thus, if the base chunk of a referenced old delta can be directly found in the restore cache during restore and thus does not require extra I/O, its fingerprint would also be directly found in the prefetched metadata during deduplication. That is, base-fragmented chunks can be identified with the assistance of the prefetched metadata during deduplication.

3.3 Fine-grained Redundancy Prohibited by Rewriting

The rewriting technique declares infrequently reused containers, which should not "share" redundant data with the current backup to alleviate chunk fragmentation. To this end, fragmented chunks need to be rewritten. To cooperate with rewriting, data chunks in the infrequently reused containers also cannot serve as base chunks for delta compression. However, existing rewriting techniques only consider duplicate chunks when identifying infrequently reused containers, without the consideration of similar chunks. In practice, infrequently reused containers may contain many similar chunks. This will lead to a significant compression loss if these similar chunks cannot serve as base chunks for delta compression.

Actually, delta compression can be considered a process involving two steps. Specifically, the first step is to generate a delta for the target chunk, which leads to two versions of the target chunk: an encoded delta and an un-encoded one. The second step is to remove the un-encoded target chunk. If the target of delta compression is changed to previously written chunks, rather than data chunks in the ongoing backup as in the traditional delta compression method, there will be two versions of a previously written chunk. If the un-encoded one can be removed during GC, delta compression benefits will be obtained from the data chunks in infrequently reused containers without affecting the efficiency of rewriting.

4 Design and Implementation

4.1 LoopDelta Overview

LoopDelta is built on a typical deduplication strategy that groups data chunks into containers and accesses containers to prefetch metadata during deduplication to accelerate duplicate detection. It aims to embed delta compression in inline deduplication for highly efficient data reduction. The key idea behind LoopDelta is the combined use of the following four key techniques:

- **Dual-locality-based Similarity Tracking.** By exploiting both the logical and physical locality based on the observations in Section 3.1, dual-locality-based similarity tracking identifies the containers that hold potential similar chunks, as detailed in Section 4.2.

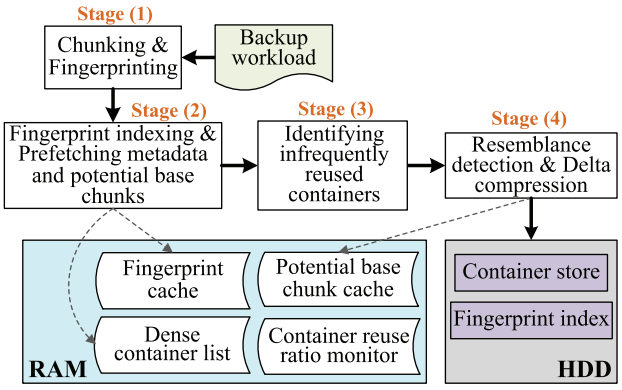


Figure 5: An overview of LoopDelta. The dashed arrows point to key data structures residing in DRAM required for the corresponding LoopDelta stages.

- **Locality-aware Prefetching.** For containers holding potential similar chunks declared by the dual-locality-based similarity tracking, when they are accessed during deduplication to prefetch metadata, data chunks are also prefetched, i.e., piggybacked on the retrieval of the metadata to serve as potential base chunks, thereby avoiding extra I/Os for reading base chunks during backup, as detailed in Section 4.3.
- **Cache-aware Filter.** LoopDelta identifies base-fragmented chunks with the assistance of recently prefetched metadata during deduplication and rewrites them to prevent extra I/Os for base chunks during restore, as detailed in Section 4.4.
- **Inversed Delta Compression.** For data chunks whose similar chunks are prefetched from infrequently reused containers, LoopDelta delta-encodes the prefetched similar chunks relative to these data chunks while deferring the removal of the data of these delta encoded chunks to GC, as detailed in Section 4.5.

The overall workflow of the LoopDelta is illustrated in Figure 5, which includes four key stages. In stage (1), the backup stream is chunked and fingerprinted. Then, duplicate chunks are identified by indexing fingerprints in stage (2). In stage (2), potential base chunks and their sketches are loaded into the potential base chunk cache.

In stage (3), a rewriting approach is adopted to identify infrequently reused containers and fragmented chunks. This stage can be skipped to disable rewriting. Note that base-fragmented chunks are also identified in stages (2) and (3), as detailed in Section 4.4. In stage (4), for all unique, fragmented, and base-fragmented chunks, LoopDelta detects their similar chunks from the potential base chunk cache and performs delta compression for them if their base chunks exist. Finally, unremoved unique chunks and deltas are appended to a container.

In LoopDelta, a container consists of a metadata section and a data section, the same as that in [17, 53]. Data chunks and deltas are stored in the data section, while their metadata such

as fingerprints, chunk length, and positions in the container are stored in the metadata section. We use a bitmap to specify which ones in the container are stored as deltas. For deltas, the metadata of their base chunks is also stored in the metadata section. For data chunks, their sketches are stored along with data chunks in the data section.

4.2 Dual-locality-based Similarity Tracking

The dual-locality-based similarity tracking is designed to identify similar chunks for delta-compressing the next backup. To capture similar chunks with logical locality, it tracks data chunks and base chunks of delta-compressed chunks of the ongoing backup. Meanwhile, to capture similar chunks with physical locality, it tracks data chunks stored in the same containers as the aforementioned data chunks.

Specifically, we define the *reuse ratio* of a container for a backup as the fraction of data chunks in this container referenced by this backup, $\frac{\textit{the recorded chunk size}}{\textit{the container size}}$, and use a *container reuse ratio monitor* to keep track of the reuse ratio of containers referenced by data chunks and base chunks of delta-compressed chunks of the on-going backup. When the backup completes, all containers holding potential similar chunks are recorded in the container reuse ratio monitor and are stored in a *dense container list*. Containers in this list will be prefetched to provide potential similar chunks for resemblance detection in the next backup.

Our approach for prefetching base chunks requires extra transfer time, thereby decreasing backup throughput, as detailed in the next subsection. To reduce this transfer time, only containers holding a large number of potential similar chunks can be recorded in the dense container list. Since containers with a larger reuse ratio are likely to contain more potential similar chunks for the next backup, we define a *dense container threshold* and only *dense containers* whose reuse ratios are greater than this threshold are included in the dense container list.

4.3 Locality-aware Prefetching

Locality-aware prefetching is designed to prefetch potential base chunks by piggybacking on routine operations for prefetching metadata during deduplication. LoopDelta adopts the duplicate detection strategy proposed by Zhu et al. [53], which employs an on-disk fingerprint index combined with an in-memory fingerprint cache and a Bloom filter for duplicate detection. Specifically, for each data chunk presented for storage, its fingerprint is compared against a fingerprint cache, and on a miss, a Bloom filter is checked to determine whether the data chunk is likely to exist in the system. If true, the on-disk fingerprint index is checked, and the metadata in the corresponding container is prefetched into the fingerprint cache. The fingerprints of the subsequent data chunks are likely to be matched in the fingerprint cache due to redundancy locality.

To prefetch potential base chunks by piggybacking on read operations for prefetching metadata, the dense container list of the last backup is loaded into memory to build a lookup table at the beginning of a backup. For each container to be accessed during deduplication, we check whether it exists in the dense container list generated by the last backup. If true, the whole container, including metadata and data chunks, is prefetched for both deduplication and delta compression; otherwise, only metadata are prefetched for deduplication. If the whole container is prefetched, all data chunks in the container as well as their sketches are inserted into the potential base chunk cache for resemblance detection. In LoopDelta, only non-delta-compressed chunks can serve as base chunks, as suggested by [37]. Thus, deltas are not loaded into the potential base chunk cache. When eviction occurs, based on the Least Recently Used (LRU) policy, data chunks and sketches from a container are evicted from the potential base chunk cache as a group.

Though locality-aware prefetching eliminates the seek and rotational delays of I/Os for reading base chunks, it increases the transfer time for prefetching data chunks in dense containers. The proposed dual-locality-based similarity tracking reduces this overhead by defining a dense container threshold, as detailed in the last subsection. Prefetching potential similar chunks according to a dense container list may be inefficient as containers in the list might have been reclaimed by GC, for which how to update the dense container list judiciously to minimize the problem will be discussed in Section 4.6. The previous technique for caching metadata on an SSD [1] is orthogonal to LoopDelta and could be used in LoopDelta to increase backup throughput.

4.4 Cache-aware Filter

Cache-aware filter is designed to identify base-fragmented chunks, which will be rewritten to achieve better restore performance. As introduced in Section 3.2.2, base-fragmented chunks can be identified with the assistance of the prefetched metadata during deduplication. Since the metadata prefetched by the routine operations during deduplication are loaded into the fingerprint cache, if the base chunk of a referenced previously written delta can be directly found in the restore cache during restore and thus does not require extra I/O, its fingerprint would also be directly found in the fingerprint cache in the fingerprint indexing stage.

When rewriting is applied, even if a base chunk's fingerprint can be directly found in the fingerprint cache, this base chunk can still trigger I/Os during restore. This is because a container whose metadata are prefetched into the fingerprint cache might be identified as an infrequently reused container by the rewriting approach in the next stage, i.e., stage (3) in Figure 5, so the data chunks it containers cannot serve as base chunks. Accordingly, the cache-aware filter adopts a two-step approach to identify base-fragmented chunks. First, it identifies the base-fragmented chunks referring to deltas whose base chunks do not exist in the fingerprint cache during

fingerprint indexing. Then, it identifies the base-fragmented chunks referencing to deltas whose base chunks exist in the infrequently reused containers.

Specifically, in the fingerprint indexing stage, if a data chunk (say, *CK*) refers to a previously-written delta (say, *Delta_old*), the fingerprint of the delta's base chunk is fetched from the fingerprint cache and compared against the fingerprint cache. If the fingerprint does not exist in the fingerprint cache, *CK* is identified as a base-fragmented chunk; otherwise, the detected container ID (say, *CID_base*) of the base chunk is associated with *CK*. In the stage of identifying infrequently reused containers, if *CK* is identified as a fragmented chunk by the rewriting approach, there is no need to identify whether it is a base-fragmented chunk because it will be rewritten; otherwise, if *CK* is not a fragmented chunk, we further check whether the container whose ID is *CID_base* is selected to avoid being deduplicated against by the rewriting approach. If true, *CK* is identified as a base-fragmented chunk; otherwise, *CK* is identified as a duplicate chunk.

4.5 Post-deduplication Delta Compression

Inversed Delta Compression. The data chunks in the infrequently reused containers cannot serve as base chunks for delta compression; otherwise, the efficiency of the rewriting technique would be reduced. Inversed delta compression is designed to exploit the benefits of delta compression prohibited by rewriting. For a new chunk (say, *N*) that has a similar chunk (say, *S*) detected from the potential base chunk cache, the traditional direct delta compression approach delta-encodes *N* relative to *S* and generates a delta (say, $Delta_{n,s}$). Then, $Delta_{n,s}$ is stored instead of *N* to achieve data reduction.

On the contrary, inversed delta compression delta-encodes *S* relative to *N* and generates a delta (say, $Delta_{s,n}$), and then *stores* $Delta_{s,n}$ *along with N*. Since inversed delta compression generates an additional encoded *S*, the un-encoded *S* in the infrequently reused container will be removed during the next GC. It should be noted that delta-decoding a delta generated by inversed delta compression usually does not require extra I/Os for reading the base chunk because the delta (e.g., $Delta_{s,n}$) is stored together with the base chunk (e.g., *N*) in the same container, except that GC may occasionally disperse them to different containers.

Inversed delta compression increases the size of the data to be stored because it needs to store additional deltas, thereby increasing the I/O overheads for writing data relative to direct delta compression. Since LoopDelta is I/O-intensive, it only performs inversed delta compression for data chunks whose base chunks are prefetched from infrequently reused containers. Besides, inversed delta compression also causes more duplicate chunks to be removed during GC, which will be discussed in Section 4.6.

Delta Compression Workflow. For each unique, fragmented, and base-fragmented chunk, LoopDelta detects its base chunk from the potential base chunk cache and performs either direct or inversed delta compression according to whether rewriting is disabled or not. If it is disabled, LoopDelta only performs direct delta compression; otherwise, data chunks in the potential base chunk cache are divided into two categories: the ones prefetched from frequently reused containers and those prefetched from infrequently reused containers. When detecting base chunks, LoopDelta prefers the data chunks prefetched from frequently reused containers. Only if no such data chunk is detected, can data chunks prefetched from infrequently reused containers, if any, be selected as base chunks.

Table 1: Workload characteristics of the tested datasets.

Name	Size	Workload descriptions	Key property
RDB	1080GB	200 backups of the redis key-value store database.	Multi-version inheritance
WEB	330GB	120 days' snapshots of the website: news.sina.com. Snapshots of each day are combined into a tar file.	Self-reference duplicate and similar chunks
CHM	284GB	100 versions of source codes of Chromium project from v84.0.4110 to v86.0.4215. Each version is combined into a tar file.	
SYN	335GB	180 versions of synthetic datasets generated by simulating file create /delete/modify operations.	Multi-version inheritance

4.6 Garbage Collection

In the GC process, for a data chunk with multiple physical instances, the system marks the most recently written one as the live chunk. The delta-encoded chunks for inversed delta compression are removed in this process. GC may reduce the efficiency of locality-aware prefetching because containers in the dense container list generated by the proposed dual-locality-based similarity tracking in the last backup might have been reclaimed. To solve the problem, we update the dense container list to track potential base chunks after GC. Note that each backup stream has only one list to be updated. Thus, compared with GC, overheads for updating the dense container lists are negligible.

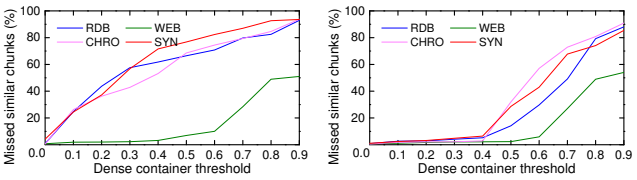
Moreover, inversed delta compression causes more duplicate chunks to be removed during GC, and the extra overhead introduced by inversed delta compression are negligible. This is because GC is time-consuming as it involves a large number of I/Os and the deduplication phase is not the bottleneck [10, 14, 17, 40].

5 Performance Evaluation

5.1 Evaluation Setup

Experimental Platform. We perform our evaluation experiments on a workstation running Ubuntu 18.04 with an Intel Xeon(R) Silver 4215R CPU @ 3.20GHz, 32GB memory, Samsung 860 PRO SSDs, and Seagate 7200RPM SATA III HDDs.

System Configurations. For all approaches under evaluation, deduplication is configured to use the Rabin-based chunking algorithm [33, 34] with the minimum, average, and maximum



(a) LoopDelta without Capping (b) LoopDelta with Capping

Figure 6: Percentage of missed similar chunks as the dense container threshold varies from 0 through 0.9 on the four datasets. The Capping's capping level in this test is 15, namely, a 20MB segment (a sequence of consecutive data chunks) refers to at most 15 containers [22].

chunk sizes of 2KB, 8KB, and 64KB respectively for chunking and the SHA1 hash function for fingerprinting. The fingerprint cache has 256 slots to hold prefetched metadata. During restore, the restore cache is configured as a 512-container-sized (2GB) LRU cache.

For delta compression, we use Odess [54] for finding similar chunks and Xdelta [25] for delta encoding. For data chunks neither deduplicated nor delta-compressed, we compress them with a local compressor called ZSTD [13] before writing them into a container, the same as in [32]. The the container size is set to 4MB. To simulate backup and restore scenarios, an HDD is used as the backup space to store ingested data, and an SSD is used as the user space to store the original datasets.

Performance Metrics. We use three metrics to evaluate the performance of LoopDelta. *The compression ratio* is used to measure the total data reduction achieved by any compression technologies, including deduplication, delta compression, and local compression. It is calculated as $\frac{original\_bytes}{post\_compression\_bytes}$, so a compression ratio of greater than 1 means data reduction.

The speed factor (MB/container-read) is defined as the mean data size restored per container read [7, 8, 22], which is used to measure the restore performance. A larger speed factors indicates better restore performance. *The backup throughput* is measured by the throughput at which the input data are deduplicated, delta compressed, and written to the disk. We run each experiment five times to obtain a stable and average value of the backup throughput. Additionally, the shown speed factor is the average of the last 20 backups and the shown backup throughput is the average of the last 10 backups.

Evaluated Datasets. Four datasets, shown in Table 1 with their key characteristics, are used for performance evaluation. These datasets represent various typical workloads, including database snapshots, website snapshots, an open-source code project, and a synthetic dataset.

5.2 A Performance Study of LoopDelta

5.2.1 Dense Container Threshold

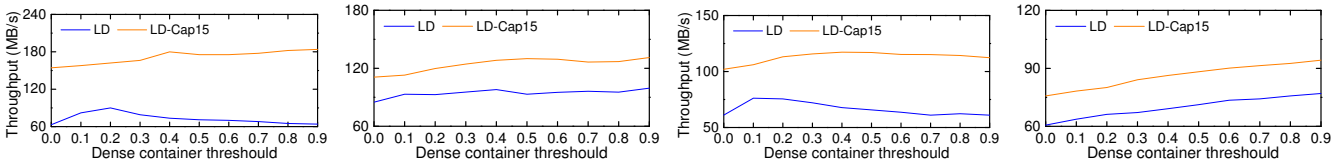
The dense container threshold can affect the number of detected similar chunks because it prevents containers whose reused ratios are smaller than it from being prefetched to supply potential similar chunks. Figure 6(a) suggests that, without rewriting, the percentage of similar chunks missed by LoopDelta increases quickly with the dense container threshold. One exception is the WEB dataset, where the percentage of missed similar chunks is low when the dense container threshold is small than 0.6. This is because this dataset contains substantial self-referenced duplicate chunks and most of the containers' reuse ratios are greater than 0.5.

Figure 6(b) suggests that Capping [22], a state-of-the-art rewriting approach introduced in Section 2.1, significantly decreases the percentage of missed similar chunks, especially when the dense container threshold is smaller than 0.4. By increasing the sequential layout of the current backup, rewriting improves both the logical and physical locality for the current and subsequent backups. This is why the percentage is low when the dense container threshold is small than 0.4. As the threshold increases beyond this, some containers holding data chunks that are inherited from the last backup are prevented from being prefetched for resemblance detection and thus the percentage of missed similar chunks grows quickly. When the dense container threshold is 0.3, LoopDelta only misses 1%-5% of similar chunks on the four datasets.

The dense container threshold can also affect backup throughput because it is related to two categories of I/O overheads: (1) transfer time for prefetching dense containers, which decreases as the threshold increases because fewer containers will be prefetched, and (2) container-writeback time saved by delta compression, which decreases as the threshold increases because fewer chunks will be delta compressed.

Figure 7 suggests that, except for the SYN dataset, the backup throughput hits a maximum and then either decreases or flattens out. This is because when the dense container threshold is very low, almost all containers holding potential similar chunks will be prefetched, leading to significant extra transfer time that exceeds the amount of container-writeback time saved by delta compression and resulting in very low throughput. This trend continues with the increase in dense container threshold until the decrease in extra transfer time for prefetching is offset by the decrease in the container-writeback time saved by delta compression. Beyond this point, the extra transfer time for prefetching becomes either greater than or equal to the decrease in container-writeback time saved by delta compression, causing the throughput to decrease or remain unchanged.

The SYN dataset contains only a few similar chunks that can be delta-compressed, and thus the container-writeback time saved by delta compression has a limited impact on backup throughput. Consequently, the backup throughput increases with the dense container threshold. Considering the two metrics (backup throughput and compression ratio) as a whole, in what follows, when rewriting (Capping) is applied, the dense container threshold is set to 0.3 as suggested by Figure 6(b) and discussed earlier.



(a) RDB (b) WEB (c) CHRO (d) SYN

Figure 7: Backup throughput as the dense container threshold varies from 0 through 0.9 on the four datasets. LD refers to LoopDelta without rewriting and LD-Cap15 refers to LoopDelta with Capping of a capping level of 15.

Table 2: Compression ratio and speed factor for LoopDelta (LD), with and without the cache-aware filter (CAF), and with and without rewriting (Capping), for the four datasets.

Dataset	Approach	Compression ratio w/o GC	Speed factor
RDB	LD w/o CAF	142.6	2.73
	LD	139.2 (-2.4%)	2.81 (+2.9%)
	LD-Cap15 w/o CAF	68.4	3.1
	LD-Cap15	60.9 (-11%)	4.67 (+50.6%)
WEB	LD w/o CAF	112.2	2.81
	LD	105.7 (-5.8%)	2.96 (+5.3%)
	LD-Cap15 w/o CAF	51.2	6.84
	LD-Cap15	46.9 (-8.4%)	7.64 (+11.7%)
CHRO	LD w/o CAF	70.7	2.66
	LD	70.3 (-0.6%)	2.73 (2.6%)
	LD-Cap15 w/o CAF	20.8	6.09
	LD-Cap15	20.4 (-1.9%)	8.12 (+33.3%)
SYN	LD w/o CAF	33.9	0.84
	LD	33.8 (-0.3%)	0.85 (+1.2%)
	LD-Cap15 w/o CAF	17.1	1.38
	LD-Cap15	15.4 (-9.9%)	2.04 (+47.8%)

5.2.2 Cache-aware Filter (CAF)

This subsection investigates the efficiency of the cache-aware filter (CAF). Since base-fragmented chunks identified by CAF will be rewritten, and rewritten data will be removed during GC. We do not run GC to show the trade-off between the decrease of compression ratio and the increase of speed factor caused by CAF.

Table 2 suggests that CAF slightly increases the speed factor (1.2%-5.3%, average of 3%) when rewriting is not applied, and it significantly increases the speed factor when rewriting is applied (by up to 50.6%, average of 35.9%), at the expense of a modest decrease (0.3%-11%, average of 4.8%) in compression ratio. CAF reduces the compression ratio because base-fragmented chunks are rewritten and not removed from storage as GC is not run. The decrease in compression ratio caused by CAF on the WEB and CHRO datasets is relatively small compared to the other two datasets. This is because these two datasets are tar-type files containing substantial small files, so most of their rewritten chunks can be delta-compressed.

A larger fingerprint cache can help to identify more base-fragmented chunks at the cost of increased computational overheads. Since the bottleneck of LoopDelta lies in I/O overheads, the computational overhead of CAF has almost no impact on backup throughput. We found that, most base-fragmented chunks can be identified by the first few slots at the front of the fingerprint cache due to locality, especially when rewriting is applied. To reduce the computational overhead, we suggest using the first 20 slots of the fingerprint cache to identify base-fragmented chunks when rewriting is applied and using the first 64 slots when rewriting is not applied. This strategy does not compromise restore performance because base-fragmented chunks that are not identified will still be recognized due to their container IDs not being found in the fingerprint cache.

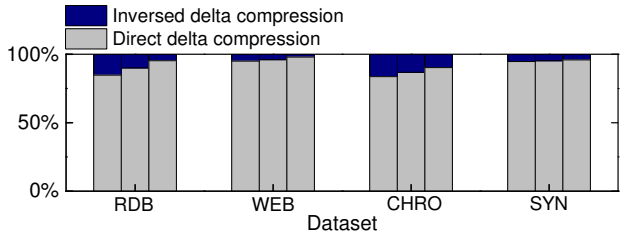
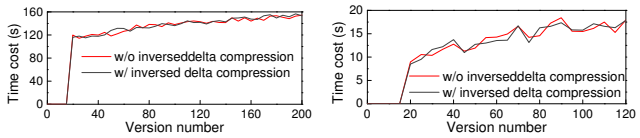


Figure 8: Proportion of compression ratio achieved by direct and inversed delta compression with different capping level on the four datasets. The three bars on each dataset from left to right represent the three capping levels of 10, 15, and 20, respectively.



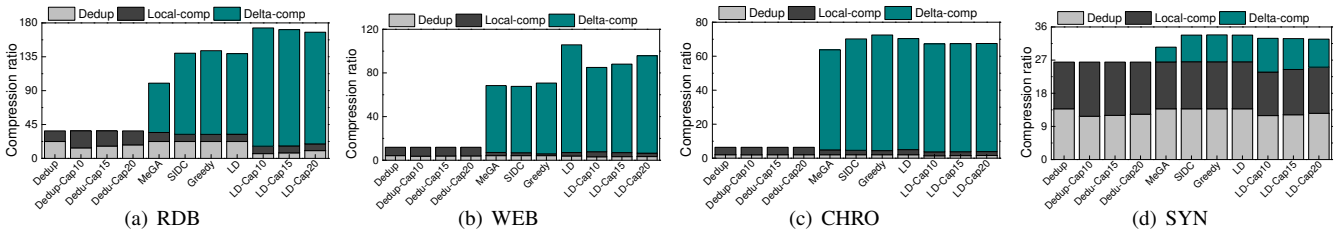
(a) RDB (b) WEB

Figure 9: Time cost of GC for LD-Cap10 with and without inversed delta compression on the RDB and WEB datasets.

5.2.3 Inversed Delta Compression

This subsection investigates the efficiency of inversed delta compression. Figure 8 suggests that the compression gains achieved by inversed delta compression account for 2.2%-16.4% of the combined compression gains by direct and inversed delta compression. For example, for LD-Cap10, the compression ratio achieved by inversed delta compression accounts for 15.3%, 5%, 16.4%, and 5.3% respectively of that achieved jointly by direct and inversed delta compression on the four datasets.

Inversed delta compression causes more data to be stored because it needs to store the deltas of data chunks being delta-encoded. For LD-Cap10, extra stored data caused by inversed delta compression account for 2.1%, 0.8%, 2.7%, and 0.1% of



(a) RDB (b) WEB (c) CHRO (d) SYN

Figure 10: Comparison of compression ratio achieved by the eleven approaches on the four datasets.

the total stored data on the four datasets respectively, which are marginal. This is because the size of a delta is often much smaller than that of a data chunk compressed with the local compressor, e.g., the former is 1/26-1/10 of the latter in size in our test.

Inversed delta compression also causes more data chunks to be removed during GC. Figure 9 compares the time cost of LD-Cap10 with and without inversed delta compression on the RDB and WEB datasets. To accumulate more deltas generated by inversed delta compression, we run GC after every 5 backups from the 20_{th} backup. The results in Figure 9 suggest that inversed delta compression has a negligible impact on the time cost of GC because the bottleneck of GC lies in marking and moving forward the live chunks, which requires a large number of I/Os.

5.3 Comprehensive Evaluation of LoopDelta

In this section, we comprehensively evaluate the performance of LoopDelta in terms of three key metrics: compression ratio, speed factor, and backup throughput. Five data reduction approaches are also tested: Dedup, Dedup-Cap, MeGA, SIDC, and Greedy. Specifically, Dedup is a deduplication approach proposed by Zhu et al. [53] without rewriting, and Dedup-Cap refers to Dedup with Capping. MeGA, SIDC, and Greedy have been discussed in Section 3.1. Dedup-Cap# and LD-Cap# represent Dedup-Cap and LD-Cap with a different capping level of #.

MeGA's restore performance and backup throughput are not evaluated because it requires additional offline and service-disruptive operations. In contrast, LoopDelta focuses on adding delta compression to inline deduplication systems in a non-service-disruptive manner. In evaluations of this section, we use a 20-container-sized base-chunk cache for LD-Cap#, and a 150-container-sized base-chunk cache for MeGA, SIDC, Greedy, and LD. With rewriting (Capping), a 20-container-sized cache can capture almost all similar chunks (not shown due to space limit).

Compression Ratio. In this test, we run GC after each backup from the 20_{th} backup. Figure 10 suggests that LD achieves a compression ratio comparable to SIDC and Greedy and higher than MeGA on the RDB, CHRO, and SYN datasets. This means that LD can detect most potential similar chunks by exploiting both logical and physical locality. On the WEB dataset, LD achieves the highest compression ratio because it can detect more highly similar chunks than the other approaches on the datasets containing self-referenced similar chunks. Specifically, LD outperforms MeGA, SIDC, and Greedy by 1.55×, 1.56×, and 1.5×, respectively, on the WEB dataset. Furthermore, LD achieves significantly higher compression ratios than Dedup, with improvements of 3.81× (RDB), 8.9× (WEB), 10.97× (CHRO), and 1.27× (SYN).

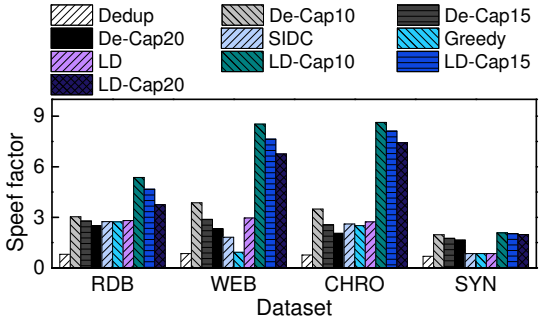


Figure 11: Comparison of speed factor achieved by the ten approaches on the four datasets.

Additionally, LD-Cap15 achieves 4.68× (RDB), 7.42× (WEB), 10.51× (CHRO), and 1.24×(SYN) higher compression ratio than Dedup-Cap15, respectively. Rewriting decreases the compression ratio of LD because a small number of similar chunks are missed, as analyzed in Section 5.2.1. One exception is the RDB dataset, where rewriting increases the compression ratio. This is because rewriting causes the rewritten chunks to be detected as base chunks. Compared to the data chunks written much earlier, rewritten chunks are more similar to the data chunks in the current backup. In our tests, the average DCE of LD and LD-Cap15 are 0.9398 and 0.9591, respectively. Note that while the total amount of redundancy eliminated by LD-Cap15 may not be significantly more than that eliminated by LD, compressing even a small amount of additional data can lead to a significant increase in compression ratio when the compression ratio is very high.

Note that LD-Cap# may achieve smaller deduplication gains and more delta compression gains than the other approaches. This is because LD-Cap# rewrite fragmented chunks and perform delta compression for them, which reduces deduplication gains but increases delta compression gains.

Speed Factor. Figure 11 suggests that LD achieves the highest speed factor among all approaches without rewriting and LD-Cap# also achieve a higher speed factor than Dedup-Cap#. Specifically, LD achieves 3.48× (RDB), 3.47× (WEB), 3.57× (CHRO), and 1.24× (SYN) higher speed factor than

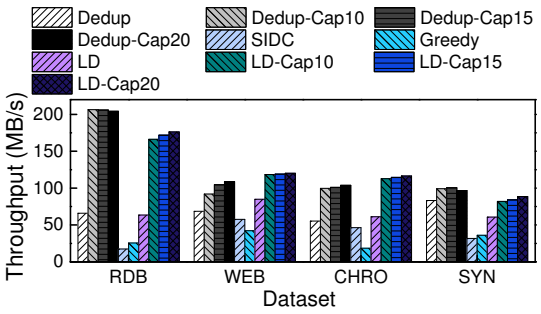


Figure 12: Comparison of backup throughput achieved by the ten approaches on the four datasets.

Dedup respectively, and LD-Cap15 achieves 1.68× (RDB), 2.65× (WEB), 3.17× (CHRO), and 1.2× (SYN) higher speed factor than Dedup-Cap15 respectively.

Generally, approaches combining deduplication and delta compression, i.e., SIDC, Greedy, and LD, achieve a higher speed factor than Dedup, which only performs deduplication. This is because delta compression has the potential to increase restore performance, which mainly depends on the number of read-in containers during restore. Delta compression decreases the number of written containers during backup and the number of read-in containers during restore. However, if base chunks require extra I/Os during restore, the improvement in speed factor resulting from delta compression will decrease. In LD, base chunks do not require extra I/Os during restore, and this is ensured by CAF.

Greedy detects similar chunks without considering the positions of base chunks. Thus, base chunks in it may require substantial I/Os during restore and this will lead to a lower speed factor, e.g., on the WEB dataset. SIDC detects similar chunks stored along with duplicate chunks, and thus, the base chunks of the delta-compressed chunks in it do not require extra I/Os during restore. However, SIDC fails to identify base-fragmented chunks. This is why LD achieves a slightly higher speed factor than SIDC on the RDB, CHRO, and SYN datasets. LD also achieves a higher speed factor than SIDC on the WEB dataset because it achieves higher compression ratio than SIDC, which means fewer read-in containers during restore.

Backup Throughput. Figure 12 suggests that LD and LD-Cap# achieve lower backup throughput than Dedup and Dedup-Cap# respectively on the RDB and SYN datasets and higher backup throughput than them on the WEB and CHRO datasets. Specifically, LD achieves 3.6% and 27.1% lower backup throughput than Dedup on the RDB and SYN datasets, and 24% and 10.5% higher backup throughput than Dedup on the WEB and CHRO datasets. Meanwhile, LD-Cap15 achieves 16.5% and 16.3% lower backup throughput than Dedup-Cap15 on the RDB and SYN datasets, and 13.9% and 13.5% higher backup throughput than Dedup-Cap15 on the WEB and CHRO datasets. Additionally, LD (LD-Cap15) achieves 1.3×∼3.7× (2.3×∼9.9×) higher backup throughput than SIDC and Greedy on the four datasets. This is because LoopDelta eliminates the seek and rotational delays of I/Os for reading base chunks.

Compared with deduplication-based backup systems, post-deduplication delta compression adds additional computational overheads, but the backup throughput is mainly decided by I/O overheads. In LoopDelta, multiple tasks in its workflow involve I/Os, including (1) looking up the fingerprint index, (2) prefetching metadata, (3) prefetching potential base chunks, (4) updating the fingerprint index, and (5) writing back containers. For datasets with low redundancy, such as the WEB and CHRO datasets, there are more unique chunks that lead to more I/Os in tasks (4) and (5), making them the performance bottleneck. Delta compression increases the backup throughput on such datasets because it alleviates the performance bottleneck by reducing I/Os in tasks (5). For datasets with high redundancy, such as the RDB and SYN datasets, there are more duplicate chunks that result in more I/Os in tasks (1), (2), and (3), making them the performance bottleneck. Delta compression decreases backup throughput because it aggravates the performance bottleneck by increasing I/Os in task (3).

To sum up, LoopDelta achieves a comparable or higher compression ratio, higher restore performance, and higher backup throughput than the other post-deduplication delta compression approaches. It also increases the compression ratio by 1.24∼10.97 times on top of deduplication, without notably affecting backup throughput, and improves the restore performance by 1.2∼3.57 times.

6 Conclusion

In this paper, we present LoopDelta to embed delta compression in inline deduplication. The key idea of LoopDelta is the combined use of four key techniques, i.e., dual-locality-based similarity tracking to detect similar chunks, locality-aware base prefetching to avoid extra I/Os for reading base chunks on the write path, cache-aware filter to avoid extra I/Os for reading base chunks on the read path, and inversed delta compression to perform delta compression for similar chunks prefetched from infrequently reused containers. The experimental results indicate that LoopDelta significantly increases compression ratio and improves restore speed over deduplication, without notably affecting backup throughput.

7 Acknowledgment

We are grateful to our shepherd and the anonymous reviewers for their insightful comments and feedback on this work. This research was partly supported by the National Natural Science Foundation of China No. 61821003, No. 61832007, No. 62262042, No. 62271239, No. 62172160, and No. 62062034; the US NSF CNS-2008835; the Jiangxi Provincial Natural Science Foundation No. 20224BAB202017, No. 20212ACB212002, and No. 20224ABC03A01; the Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2021WNLOKF012.

References

[1] Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale. Can't we all get along? redesigning protection storage for modern workloads. In *the 2016 conference on USENIX Annual Technical Conference (ATC'18*, pages 705–718, Boston, MA, USA, July 11 - 13 2018. USENIX Association.

[2] Yamini Allu, Fred Douglis, Mahesh Kamat, Philip Shilane, Hugo Patterson, and Ben Zhu. Backup to the future: How workload and hardware changes continually redefine data domain file systems. *Computer*, 50(7):64–72, 2017.

[3] George Amvrosiadis and Medha Bhadkamkar. Identifying trends in enterprise data protection systems. In *the 2015 conference on USENIX Annual Technical Conference (ATC'15)*, pages 151–164, Santa Clara, CA, July 08 - 10 2015. USENIX Association.

[4] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T Klein. The design of a similarity based deduplication system. In *the 2th Annual International Systems and Storage Conference (SYSTOR'09)*, pages 1–14, Haifa, Israe, May 04 - 06 2009. ACM Association.

[5] Fabiano C Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. Memory efficient sanitization of a deduplicated storage system. In *the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 81–94, San Jose, CA, February 12 - 15 2013. USENIX Association.

[6] Andrei Z Broder. Identifying and filtering near-duplicate documents. In *Annual Symposium on Combinatorial Pattern Matching*, pages 1–10. Springer, 2000.

[7] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David HC Du. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, pages 129–142, Boston, MA, USA, February 25 - 28 2019. USENIX Association.

[8] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 309–324, Oakland, CA, USA, February 12 - 15 2018. USENIX Association.

[9] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *the 2010 conference on USENIX Annual Technical Conference (ATC'10)*, pages 1–16, Boston, MA, USA, June 23 - 25 2010. USENIX Association.

[10] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 29–44, Santa Clara, CA, USA, February 12 - 15 2017. USENIX Association.

[11] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *the 2003 USENIX conference on USENIX Annual Technical Conference (ATC'03)*, pages 113–126, San Antonio, TX, USA, June 09 - 14 2003. USENIX Association.

[12] Kruus Erik, Ungureanu Cristian, and Dubnicki Cezary. Bimodal Content Defined Chunking for Backup Streams. In *the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, pages 1–14, San Jose, CA, USA, February 23 - 26 2010. USENIX Association.

[13] Facebook. Zstandard. https://github.com/facebook/zstd, September 2022. zstd.

[14] Min Fu, Dan Feng, Yu Hua, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *the 2014 USENIX conference on USENIX Annual Technical Conference (ATC'14)*, pages 181–192, Philadelphia, PA, USA, June 19 - 20 2014. USENIX Association.

[15] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):855–868, 2015.

[16] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 331–345, Santa Clara, CA, USA, February 16 - 19 2015. USENIX Association.

[17] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *the 2011 USENIX conference on USENIX Annual Technical Conference (ATC'11)*, pages 1–14, Portland, OR, USA, June 15 - 17 2011. USENIX Association.

[18] Diwaker Gupta, Sangmin Lee, Michael Vrable, and et al. Difference engine: Harnessing memory redundancy in virtual machines. In *the 5th Symposium on Operating Systems Design and Implementation (OSDI'08)*, pages 309–322, San Diego, CA, USA, December 08 - 10 2008. USENIX Association.

[19] Navendu Jain, Michael Dahlin, and Renu Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *the 3th USENIX Conference on File and Storage Technologies (FAST'05)*, pages 281–294, San Francisco, CA, USA, March 13 - 16 2005. USENIX Association.

[20] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *the 5th Annual International Systems and Storage Conference (SYSTOR'12)*, pages 1–12, Haifa, Israe, June 04 - 06 2012. ACM Association.

[21] Purushottam Kulkarni, Fred Douglis, Jason D LaVoie, and John M Tracey. Redundancy elimination within large collections of files. In *the 2004 USENIX Annual Technical Conference (ATC'04)*, pages 59–72, Boston, MA, USA, June 27 - July 01 2004. USENIX Association.

[22] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 183–197, San Jose, CA, USA, February 12 - 15 2013. USENIX Association.

[23] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, volume 9, pages 111–123, San Jose, CA, February 24 - 27 2009. USENIX Association.

[24] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In *the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 257–271, Santa Clara, CA, USA, February 17 - 20 2014. USENIX Association.

[25] Josh MacDonald. *File system support for delta compression*. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[26] Dirk Meister and Andre Brinkmann. dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD). In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–6, Incline Village, Nevada, USA, May 03 - 07 2010. IEEE Computer Society Press.

[27] Dirk Meister, Jürgen Kaiser, and André Brinkmann. Block locality caching for data deduplication. In *the 6th International Systems and Storage Conference (SYSTOR'13)*, pages 1–12, Haifa, Israel, June 30 - July 02 2013. ACM Association.

[28] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, pages 229–241, San Jose, CA, USA, February 15 - 17 2011. USENIX Association.

[29] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A Low-Bandwidth Network File System. In *the ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 1–14, Banff, Canada, October 21 - 24 2001. ACM Association.

[30] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC'11)*, pages 581–586, Banff, Canada, September 02 - 04 2011. IEEE Computer Society Press.

[31] Fan Ni and Song Jiang. RapidCDC: Leveraging duplicate locality to accelerate chunking in CDC-based deduplication systems. In *the 10th ACM Symposium on Cloud Computing (SoCC'19)*, pages 220–232, Santa Cruz, CA, USA, 2019. ACM Association.

[32] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. Deepsketch: A new machine learning-based reference search technique for post-deduplication delta compression. In *the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, pages 247–264, Santa Clara, CA, USA, February 22 - 24 2022. USENIX Association.

[33] Sean Quinlan and Sean Dorward. Venti: a New Approach to Archival Storage. In *the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, pages 89–101, Monterey, CA, USA, January 28 - 30 2002. USENIX Association.

[34] Michael O Rabin. *Fingerprinting by Random Polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[35] B. Romański, Ł. Heldt, W. Kilian, K. Lichota, and C. Dubnicki. Anchor-driven subchunk deduplication. In

The 4th Annual International Systems and Storage Conference (SYSTOR'11), pages 1–13, Haifa, Israel, May 30 - June 01 2011. ACM Association.

[36] Shruti Sanadhya, Raghupathy Sivakumar, Kyu-Han Kim, Paul Congdon, Sriram Lakshmanan, and Jatinder Pal Singh. Asymmetric caching: Improved network deduplication for mobile devices. In *the 18th annual international conference on Mobile computing and networking (MobiCom'12)*, pages 161–172, Istanbul, Turkey, August 22 - 26 2012. ACM Association.

[37] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. WAN optimized replication of backup datasets using stream-informed delta compression. In *the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, pages 49–63, San Jose, CA, USA, 2012. USENIX Association.

[38] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta Compressed and Deduplicated Storage Using Stream-Informed Locality. In *the 4th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'12)*, Boston, MA, USA, June 13 - 14 2012. USENIX Association.

[39] Yujuan Tan, Jian Wen, Zhichao Yan, Hong Jiang, Srisaan Witawas, Baiping Wang, and Hao Luo. Fgdefrag: A fine-grained defragmentation approach to improve restore performance. In *the 33th Symposium on Mass Storage Systems and Technologies (MSST'17)*, Santa Clara, California, May 15 - 19 2017. IEEE Computer Society Press.

[40] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Cumulus: Filesystem backup to the cloud. In *the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 225–238, Santa Clara, CA, USA, February 24 - 27 2009. USENIX Association.

[41] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, pages 1–14, San Jose, CA, February 14 - 17 2012. USENIX Association.

[42] Chunzhi Wang, Yanlin Fu, Junyi Yan, Xinyun Wu, Yucheng Zhang, Huiling Xia, and Ye Yuan. A cost-efficient resemblance detection scheme for post-deduplication delta compression in backup systems. *Concurrency and Computation: Practice and Experience*, 2021.

[43] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[44] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *the 2011 conference on USENIX Annual Technical Conference (ATC'11)*, pages 285–298, Portland, OR, June 15 - 17 2011. USENIX Association.

[45] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. Edelta: A word-enlarging based fast delta compression approach. In *the 7th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'15)*, Santa Clara, CA, July 06 - 07 2015. USENIX Association.

[46] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *the 2016 conference on USENIX Annual Technical Conference (ATC'16)*, pages 101–114, Denver, CO, June 15 - 17 2016. USENIX Association.

[47] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. Online deduplication for databases. In *ACM International Conference on Management of Data (SIGMOD'17)*, pages 1355–1368, Chicago, IL, USA, 2017. ACM Association.

[48] Lawrence L You, Kristal T Pollack, and Darrell DE Long. Deep store: An archival storage system architecture. In *the 21st International Conference on Data Engineering (ICDE'05)*, pages 804–815, Tokyo, Japan, April 5 - 8 2005. IEEE.

[49] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *the 34th IEEE International Conference on Computer Communications (INFOCOM'15)*, pages 1337–1345, Hong Kong, China, April 26th - May 1st, 2015. IEEE.

[50] Yucheng Zhang, Hong Jiang, Mengtian Shi, Chunzhi Wang, Nan Jiang, and Xinyun Wu. A high-performance post-deduplication delta compression scheme for packed datasets. In *IEEE 39th International Conference on Computer Design (ICCD'21)*, pages 464–471. IEEE, October 24 - 27 2021.

[51] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *the 17th USENIX Conference on File and Storage Technologies (FAST'19)*,

pages 121–128, Boston, MA, USA, February 25 - 28 2019. USENIX Association.

[52] Yucheng Zhang, Ye Yuan, Dan Feng, Chunzhi Wang, Xinyun Wu, Lingyu Yan, Deng Pan, and Shuanghong Wang. Improving restore performance for in-line backup system combining deduplication and delta compression. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2302–2314, 2020.

[53] Benjamin Zhu, Kai Li, and Patterson Hugo. Avoiding the disk bottleneck in the data domain deduplication file system. In *the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, pages 269–282, San Jose, CA, USA, February 26 - 29 2008. USENIX Association.

[54] Xiangyu Zou, Cai Deng, Wen Xia, Philip Shilane, Haoliang Tan, Haijun Zhang, and Xuan Wang. Odess: Speeding up resemblance detection for redundancy elimination by fast content-defined sampling. In *the 37th International Conference on Data Engineering (ICDE'21)*, pages 480–491. IEEE, April 19 - 22 2021.

[55] Xiangyu Zou, Wen Xia, Philip Shilane, Haijun Zhang, and Xuan Wang. Building a high-performance fine-grained deduplication framework for backup storage with high deduplication ratio. In *the 2022 USENIX Annual Technical Conference (ATC'22)*, pages 19–36, Carlsbad, CA, USA, July 11 - 13 2022. USENIX Association.

[56] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. The dilemma between deduplication and locality: Can both be achieved? In *the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 171–185. USENIX Association, February 23 - 25 2021.

TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs

Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding
University of California, Santa Barbara

Abstract

Recently, graph neural networks (GNNs), as the backbone of graph-based machine learning, demonstrate great success in various domains (*e.g.*, e-commerce). However, the performance of GNNs is usually unsatisfactory due to the highly sparse and irregular graph-based operations. To this end, we propose **TC-GNN**, the first GNN acceleration framework based on GPU Tensor Core Units (TCUs). The core idea is to reconcile the "Sparse" GNN computation with the high-performance "Dense" TCUs. Specifically, we conduct an in-depth analysis of the sparse operations in mainstream GNN computing frameworks. We introduce a novel sparse graph translation technique to facilitate TCU processing of the sparse GNN workload. We implement an effective CUDA core and TCU collaboration design to fully utilize GPU resources. We integrate TC-GNN with the PyTorch framework for high programmability. Rigorous experiments show an average of 1.70× speedup over the state-of-the-art DGL framework across various models and datasets.

1 Introduction

Over the recent years, with the increasing popularity of graph-based learning, graph neural networks (GNNs) [27, 51, 59] become dominant in the computing of essential tasks across a wide range of domains, like e-commerce, financial services, and etc. Compared with standard methods for graph analytics, such as random walk [18, 22, 50] and graph laplacians [8, 32, 33], GNNs highlight themselves with significantly higher accuracy [27, 54, 59] and better generality [19]. From the computation perspective, GNNs feature an interleaved execution phase of both graph operations (scatter-and-gather [17]) at the ***Aggregation*** phase and Neural Network (NN) operations (matrix multiplication) at the ***Update*** phase. Our experimental studies further show that the aggregation phase which involves highly sparse computation on irregular input graphs generally takes more than 80% of the running time for both GNN training and inference. Existing GNN frameworks, *e.g.*, Deep Graph Library [55] and PyTorch Geometric [13], are mostly built upon the popular NN frameworks that are originally optimized for dense operations, such as general matrix-matrix multiplication (GEMM). To support sparse computations in GNNs, their common strategy is to incorporate sparse primitives (such as cuSPARSE [38]) for their backend implementations. However, cuSPARSE leverages the sparse linear algebra (LA) algorithm which involves lots of high-cost indirect memory accesses on non-zero elements of a sparse matrix. Therefore, cuSPARSE cannot enjoy the same level of optimizations (e.g., data reuse) as its dense counterpart, such as cuBLAS [40]. Moreover, cuSPARSE is designed to only utilize CUDA cores. Therefore, It cannot benefit from advancements in GPU hardware features, like Tensor Core Units (TCUs) on the recent NVIDIA Ampere and Hopper GPUs. Such a design is also the trend of many other AI-tailored accelerators/units (e.g., Google TPU [24] and Matrix Core [2] on AMD GPUs) and can significantly boost the performance of dense LA algorithms (e.g., GEMM and Convolution) in most conventional deep-learning applications (e.g., CV [20] and NLP [10]).

This work focuses on exploring the potential of TCUs for accelerating such GNN-based graph learning and our design/optimization principles will also benefit other similar AI hardware [2, 24] for sparse deep-learning workloads. We remark that making TCUs effective for general GNN computing is a non-trivial task. Our initial study shows that naively applying the TCU to sparse GNN computation would even result in inferior performance compared with the existing sparse implementations on CUDA cores. There are several challenges. ***First***, directly resolving the sparse GNN computing problem with the pure dense GEMM solution is impractical due to the extremely large memory cost ($O(N^2)$, where N is the number of nodes). Besides, traversing the matrix tiles already known to be filled with all-zero elements would cause excessive unnecessary computations and memory access. ***Second***, simply employing TCUs to process non-zero matrix tiles of the sparse graph adjacency matrix would still waste most of the TCU computation and memory access efforts. This is because TCU

input matrix tiles are defined with fixed dimension settings (*e.g.*, $height(16) \times width(8)$), whereas the non-zero elements of a sparse graph adjacency matrix are distributed irregularly. Thus, it requires intensive zero-value padding to satisfy such a rigid input constraint. ***Third***, although the recent CUDA release update enables TCUs to exploit the benefit of certain types of sparsity [37], it only supports blocked SpMM, where non-zero elements must first fit into well-shaped blocks and the number of blocks must be the same across different rows. Such an input restriction makes it hard to handle highly irregular sparse graphs in real-world GNN applications.

To this end, we introduce, **TC-GNN**[1], the first TCU-based GNN acceleration design on GPUs. Our key insight is to *let the sparse input graph fit the dense computation of TCUs*. ***At the input level***, instead of exhaustively traversing all sparse matrix tiles and determining whether to process each tile, we develop a new *sparse graph translation* (SGT) technique that can effectively identify those non-zero tiles and condense non-zero elements from these tiles into fewer number of "dense" tiles. Our major observation is that neighbor sharing is very common among nodes in real-world graphs. Therefore, applying SGT can effectively merge the unnecessary data loading of the shared neighbors among different nodes to avoid high-cost memory access. SGT is generic to any kind of sparse pattern of input graphs and can always yield the correct results as the original sparse algorithm. ***At the GPU kernel level***, for efficiently processing GNN sparse workloads, TC-GNN exploits the benefits of CUDA core and TCU collaboration. The major design idea is that the CUDA core, which is more powerful at fine-grained thread-level execution, would be a good candidate for managing memory-intensive data access. While TCU, which is more powerful in handling simple arithmetic operations (*e.g.*, multiplication and addition), would be well-suited for compute-intensive GEMM on dense tiles generated from SGT. ***At the framework level***, we integrate TC-GNN with the popular PyTorch [49] framework. Thereby, users only need to interact with their familiar PyTorch programming environment by using TC-GNN APIs. This can significantly reduce extra learning efforts, and improve user productivity and code portability.

To sum up, we summarize our contributions as follows:

- We conduct a detailed analysis (§3) of existing solutions (*e.g.*, SpMM on CUDA cores) and identify the potentials of TCUs for accelerating sparse GNN workloads.
- We introduce a sparse graph translation technique (§4.1). It can make the sparse and irregular GNN input graphs easily fit the dense computing of TCUs for acceleration.
- We build a TCU-tailored algorithm (§4.2) and GPU kernel design (§4.3) for CUDA core and TCU collaboration on GPUs to handle different sparse GNN computation.

[1] https://github.com/YukeWang96/TC-GNN_ATC23.git

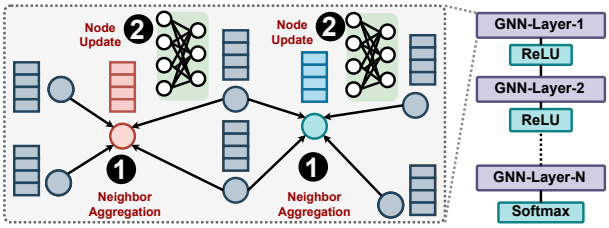


Figure 1: GNN General Computation Flow.

- Extensive experiments show TC-GNN achieves 1.70× speedup on average over the state-of-the-art GNN computing framework, Deep Graph Library, across various mainstream GNN models and dataset settings.

2 Background

2.1 Graph Neural Networks

Graph neural networks (GNNs) are an effective tool for graph-based machine learning. The detailed computing flow of GNNs is illustrated in Figure 1. GNNs basically compute the node feature vector (embedding) for node v at layer $k+1$ based on the embedding information at layer k ($k \geq 0$), as shown in Equation 1,

$$\begin{aligned} a_v^{(k+1)} &= Aggregate^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)}) \\ h_v^{(k+1)} &= Update^{(k+1)}(a_v^{(k+1)}) \end{aligned} \tag{1}$$

where $h_v^{(k)}$ is the embedding vector for node v at layer k; $a_v^{(k+1)}$ is the aggregation results through collecting neighbors' information (*e.g.*, node embeddings); $\mathbf{N}(v)$ is the neighbor set of node v. The aggregation method and the order of aggregation and update could vary across different GNNs. Some methods [19, 27] just rely on the neighboring nodes while others [54] also leverage the edge properties that are computed by applying vector dot-product between source and destination node embeddings. The update function is generally composed of standard NN operations, such as a fully connected layer or a multi-layer perceptron (MLP) in the form of $w \cdot a_v^{(k+1)} + b$, where w and b are the weight and bias parameters, respectively. The common choices for node embedding dimensions are 16, 64, and 128, and the embedding dimension may change across different layers. After several iterations of aggregation and update (*i.e.*, several GNN layers), we will get the output feature embedding of each node, which can be used for various downstream graph learning tasks, such as node classification [11, 16, 25] and link prediction [6, 28, 53].

The sparse computing in the aggregation phase is generally formalized as the sparse-matrix dense-matrix multiplication (SpMM), as illustrated in Figure 2a, and is handled by many sparse libraries (*e.g.*, cuSPARSE [38]) in many state-of-the-art GNN frameworks [55, 57]. These designs only count on GPU CUDA cores for computing, which waste the modern GPUs

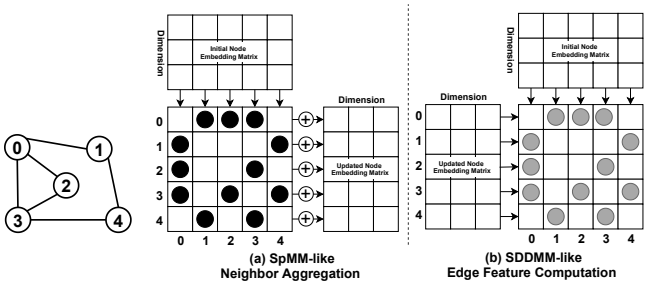


Figure 2: (a) SpMM-like and (b) SDDMM-like Operation in GNNs. Note that "→" indicates loading data; "⊕" indicates neighbor embedding accumulation.

with diverse computing units, such as the Tensor Core Unit (TCU). Specifically, we formalized the neighbor aggregation as SpMM-like operations (Equation 2)

$$\hat{\mathbf{X}} = (\mathbf{F}_{N\times N} \odot \mathbf{A}_{N\times N}) \cdot X_{N\times D}) \tag{2}$$

where $\mathbf{A}$ is the graph adjacency matrix stored in CSR format. $\mathbf{X}$ is a node feature embedding matrix stored in dense format. N is the number of nodes in the graph, and D is the size of node feature embedding dimension; $\odot$ is the elementwise multiplication and $\cdot$ is the standard matrix-matrix multiplication; $\mathbf{F}$ is the edge feature matrix in CSR format and can be computed by Sampled Dense-Dense Matrix Multiplication (SDDMM)-like operations (Equation 3 and Figure 2b).

$$\mathbf{F} = (\mathbf{X}_{N\times D} \cdot \mathbf{X}^T_{N\times D}) \odot \mathbf{A}_{N\times N} \tag{3}$$

Note that the computation of F is optional in GNNs, which is generally adopted by the Attention-based Graph Neural Network in PyTorch [51] for identifying more complicated graph structural information. Other GNNs, such as the Graph Convolutional Network [27] and Graph Isomorphism Network [59], only use the adjacency matrix for neighbor aggregation.

2.2 GPU Tensor Core

In the most recent GPU architectures (since Volta [43]), NVIDIA announced a new type of computing unit, Tensor Core Unit (TCU), for accelerating dense deep-learning operations (*e.g.*, Dense GEMM). A GPU Streaming-Multiprocessor (w/ TCU) is illustrated in Figure 3. Note that FP64, FP32, INT, and SFU are for double-precision, single-precision, integer, and special function units, respectively. Different from scalar computation on CUDA cores, TCU provides tile-based matrix-matrix computation primitives on register fragments, which can deliver more than 10× throughput improvement. In particular, TCU supports the compute primitive of $\mathbf{D} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$, where $\mathbf{A}$ and $\mathbf{B}$ are required to be a certain type of precision (*e.g.*, half, TF-32), while $\mathbf{C}$ and $\mathbf{D}$ are stored in FP32. Depending on the data precision and GPU architecture version, the matrix size (MMA shape) of $\mathbf{A}(M \times K)$, $\mathbf{B}(K \times N)$, and $\mathbf{C}(M \times N)$ should follow some principles [41]. For example, TF-32 TCU computing requires $M = N = 16$ and $K = 8$.

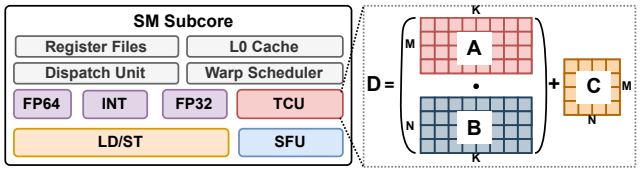


Figure 3: A Subcore of GPU SM with TCUs.

Listing 1: WMMA APIs for TCUs in CUDA C.

```
wmma::fragment<matrix_a, M, N, K, tf32, row_major> a_frag;
// Load tiles (global/shared mem. -> register fragments).
wmma::load_matrix_sync(a_frag, A, M);
// Execute GEMM on loaded tiles on register fragments.
wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
// Move results (register fragments -> global/shared mem).
wmma::store_matrix_sync(C, c_frag, N, mem_row_major);
```

In the recent CUDA release (>=11.0) on Ampere (*sm*>=80), TF-32 serves as a good alternative to float/double on TCU-based GPU computing for modern deep-learning applications, according to NVIDIA's in-depth studies [45].

Different from the CUDA cores that operate at the thread level (*e.g.*, allowing the "if" branch among threads), TCU supports only the operation at the warp level (*e.g.*, forbidding the "if" branch among threads within a warp). Before calling TCUs, all registers in a warp need to collaboratively store matrix tiles into a new memory hierarchy *Fragment* [48], which allows data sharing across registers. This intra-warp sharing provides opportunities for fragment-based memory optimizations. TCU can be utilized in several ways. The simplest way is to call cuBLAS [40] by using the `cublasSgemmEX` API. The second way is to call the Warp Matrix Multiply-Accumulate (WMMA) (`nvcuda::wmma`) API [47] in CUDA C to operate TCUs directly with four major operations (Listing 1).

Since the appearance of the TCU, research efforts have been devoted to accelerating deep-learning (DL) workloads with TCUs. Ang and Simon [31] leverage 1-bit GEMM capability on Turing TCUs for accelerating binary Neural Network inference. Boyuan et al. [12] introduce GEMM-based scientific computing on TCUs with extended precision and high performance. Yuke et al. [56] treat batched quantized GNNs (partitioning large graphs into small graphs as batches) as batched dense GEMM computation and accelerate it on TCUs for inference. These prior efforts use TCUs in the dense DL applications that TCU is initially designed for, while TC-GNN jumps out of the scope defined by TCU designers and accelerates the sparse full-graph GNNs using TCUs.

3 Motivation

In this section, we will discuss the major technical thrust for us to leverage TCUs for accelerating sparse GNN computation. We use the optimization of SpMM as the major example in this discussion, and the acceleration of SDDMM would also benefit from similar optimization principles.

Table 1: Profiling of GCN Sparse Operations.

Dataset	Aggr. (%)	Update (%)	Cache(%)	Occ.(%)
Cora	88.56	11.44	37.22	15.06
Citeseer	86.52	13.47	38.18	15.19
Pubmed	94.39	5.55	37.22	16.24

Table 2: Medium-size Graphs in GNNs.

Dataset	# Nodes	# Edges	Memory	Eff.Comp
OVCR-8H	1,890,931	3,946,402	14302.48 GB	0.36%
Yeast	1,714,644	3,636,546	11760.02 GB	0.32%
DD	334,925	1,686,092	448.70 GB	0.03%

3.1 SpMM on CUDA cores

As the major component of sparse linear algebra operation, SpMM has been incorporated in many off-the-shelf libraries [1, 3, 5, 21, 38]. The close-sourced cuSPARSE [38] library developed by NVIDIA is the most popular solution and it can deliver state-of-the-art performance for most GPU-based SpMM computation. cuSPARSE has also been widely adopted by many GNN frameworks, such as Deep Graph Library (DGL) [55], as the backend for sparse operations. To understand its characters, we profile DGL on one layer of a GCN [27] model (*neighbor aggregation + node update*) on NVIDIA RTX3090. We report two key kernel matrices for only neighbor aggregation kernel, including L1/texture cache hit rate (*Cache*) and the achieved Streaming-Multiprocessor (SM) occupancy (*Occ.*). We select three representative GNN datasets: Cora with 3,327 nodes, 9,464 edges, and 3,703 node embedding dimensions; Citeseer with 2,708 nodes, 10,858 edges, and 1,433 dimensions; Pubmed with 19,717 nodes, 88,676 edges, and 500 dimensions.

From Table 1, we have several observations: ***First***, the aggregation phase usually dominates the overall execution of the GNN execution. From these three commonly used GNN datasets, we can see that the aggregation phase usually takes more than 80% of the overall execution time, which demonstrates the key performance bottleneck of the GNNs is to improve the performance of the sparse neighbor aggregation. ***Second***, sparse operations in GNNs show very low memory performance. The column *Cache* of Table 1 shows GNN sparse operations could not well benefit from the GPU cache system, thus, showing a low cache-hit ratio (around 37%) and frequent global memory access. ***Third***, sparse operations of GNNs show very inefficient computation. As described in the column *Occupancy* of Table 1, the sparse operation of GNNs could hardly keep the GPU busy because 1) its low computation intensity (the number of non-zero elements in the sparse matrix is generally small); 2) its highly irregular memory access for fetching rows of the dense matrix during the computation, resulting in memory-bound computation; 3) it currently can only leverage CUDA cores for computation, which naturally has limited throughput performance. On the other side, this study also points out several potential directions for improving the SpMM performance on GPUs, such as improving the computation intensity (*e.g.*, assigning more workload to each thread/warp/block), boosting memory access efficiency (*e.g.*, crafting specialized memory layout for coalesced memory access), and breaking the computation performance ceiling (*e.g.*, using TCUs).

3.2 Dense GEMM on CUDA Cores/TCUs

While the dense GEMM is mainly utilized for dense NN computation (*e.g.*, linear transformation and convolution), it can also be leveraged for GNN aggregation under some circumstances. For example, when an input graph has a very limited number of nodes, we can directly use the dense adjacency matrix of the graph and accelerate the intrinsically sparse neighbor aggregation computation on CUDA cores/TCUs by calling cuBLAS [40]. However, such an assumption may not hold even for medium-size graphs in real-world GNNs.

As shown in Table 2, for these selected datasets, the memory consumption of their dense graph adjacent matrix (as a 2D `float` array) would easily exceed the device memory constraint of today's GPU (less than 100GB). Even if we assume the dense adjacent matrix can fit into the GPU memory, the extremely low effective computation (the last column of Table 2) would also be a major obstacle for us to achieve high performance. We measure the effective computation as $\frac{nnz}{N \times N}$, where nnz is the number of the non-zero elements (indicating edges) in the graph adjacent matrix and N is the number of nodes in the graph. The number of nnz is tiny in comparison with the $N \times N$. Therefore, computation and memory access on zero elements are wasted.

3.3 Hybrid Sparse-Dense Solution

Another type of work [29, 37] takes the path of mixing the *sparse control* (tile-based iteration) with *Dense GEMM computation*. They first apply a convolution-like (2D sliding window) operation on the adjacent matrix and traverse all possible dense tiles that contain non-zero elements. Then, for all identified non-zero tiles, they invoke GEMM on CUDA cores/TCUs for computation. However, this strategy has two shortcomings. ***First***, the sparse control itself would cause a high overhead. Based on our empirical study, the non-zero elements are highly scattered on the adjacent matrix of a sparse graph. Therefore, traversing all blocks in a super large adjacent matrix would be time-consuming. ***Second***, the identified sparse tiles would still waste lots of computation. The irregular edge connections of the real-world graphs could hardly fit into these fixed-shape tile frames. Therefore, most of the dense tiles would still have very few non-zero elements.

Inspired by the above studies, we make several design choices in order to achieve high-performance sparse GNN operations. ***First***, we choose the hybrid sparse-dense solution as the starting point. This can give us more flexibility for optimizations at the sparse control (*e.g.*, traversing fewer tiles)

Table 3: Comparison among Sparse GEMM, Dense GEMM, Hybrid Sparse-Dense, and TC-GNN. Note that **MC**: Memory Consumption, **EM**: Effective Memory Access, **CI**: Computation Intensity, **EC**: Effective Computation.

Solution	MC	EM	CI	EC
Sparse GEMM (§3.1)	Low	Low	Low	High
Dense GEMM (§3.2)	High	High	High	Low
Hybrid Sparse-Dense (§3.3)	High	Low	Low	High
TC-GNN (This work)	**Low**	**High**	**High**	**High**

and dense computation (*e.g.*, increasing the effective computation/memory access when processing each tile). ***Second***, we employ shared memory as the key space for GPU kernel-level data management. It can help us to re-organize the irregular GNN input data in a more "regularized" way such that both the memory access efficiency and computing performance can be well improved. ***Third***, we choose TCUs as our major computing unit since they can bring significantly higher computing throughput performance in comparison with CUDA cores. This also indicates the great potential of using TCUs for harvesting more performance gains.

Finally, we crystallize all of our ideas and insights into TC-GNN that effectively coordinates the execution of GNN sparse operations on dense TCU. We show a brief qualitative comparison among TC-GNN and the above three solutions in Table 3. Note that *Memory Consumption* is the size of memory used by the sparse/dense graph adjacency matrix; The *Effective Memory Access* is the ratio between the size of the accessed data that is actually involved in the later computation and the total data being accessed; The *Computation Intensity* is the ratio of computing operations versus the data being accessed; The *Effective Computation* is the operations for generating the final result versus the total operations.

4 TC-GNN Design

In this section, we will first give an overview of TC-GNN through its high-level programming interface and then detail the TCU-aware GNN algorithm design. As detailed in Listing 2, TC-GNN consists of several key components to facilitate the programming of GNN models on GPU TCUs. TC-GNN introduces a set of pre-built popular GNN layers (e.g., `TCGNN.GCNConv`) that can be easily connected with some other existing neural network layers (e.g., `ReLU` and `softmax`), to help users define their own GNN model quickly. For those non-conventional GNN layers, users can directly use our low-level APIs (e.g., `TCGNN.spmm` and `TCGNN.sddmm`) to express the GNN computation easily. TC-GNN introduces an input **`Loader`** to load the GNN input graph as a *rawGraph* and capture the key input information for system-level optimizations. TC-GNN incorporates a **`Preprocessor`** to build tiles from *rawGraph* and generate TCU-aware *tiledGraph* (§4.1), and optimize runtime configuration (e.g., warps per block) for our TCU-tailored GPU kernel (§4.2 and §4.3) based on input. Finally, we train the initialized GNN model defined in TC-GNN as the regular GNN models defined in other frameworks through forward and backward computation.

Listing 2: Example of a 2-layer GCN in TC-GNN.

```
import TCGNN, torch
# include other packages ...
class GCN(torch.nn.Module):
    def __init__(self, inDim, hiDim, outDim):
        self.layer1 = TCGNN.GCNConv(inDim, hiDim)
        self.layer2 = TCGNN.GCNConv(hiDim, outDim)
        self.softmax = torch.nn.Softmax()

    def forward(self, tiledGraph, param):
        tiled_adj, X = tiledGraph.adj, tiledGraph.X
        X = self.layer1(X, tiledAdj, param)
        X = self.ReLU(X)
        X = self.layer2(X, tiledAdj, param)
        X = self.softmax(X)
        return X
# Define a two-layer GCN model in TC-GNN.
model = GCN(inDim=100, hiDim=16, outDim=10)
# Load graph and extract input information.
rawGraph, info = TCGNN.Loader(graphFilePath)
# Generate TCU tile and runtime configuration.
tiledGraph, config = TCGNN.Preprocessor(rawGraph, info)
# Run model through forward computation.
predict_y = model(tiledGraph, config)
# Compute loss and accuracy.
# Gradient backpropagation for training.
```

4.1 TCU-aware Sparse Graph Translation

As the major component of TC-GNN, we introduce a novel *Sparse Graph Translation* (SGT) technique to facilitate the TCU acceleration of GNNs. Our core idea is that *the pattern of the graph sparsity can be well-tuned for TCU computation through effective graph structural manipulation meanwhile guaranteeing output correctness*. Our key observation is that neighbor sharing is common in real-world graphs and has been exploited for various tasks like link prediction [63]. Our evaluated datasets (Section 5) have 18% to 47% (averaged 29%) neighbor similarity. Specifically, we condense (remap) the highly-scattered neighbor ids into highly-condensed new neighbor ids that can facilitate the dense TCU computation paradigm. Also, such condensing should not compromise any original information (*e.g.*, edge connections) and can generate the exact output as the conventional design.

As exemplified in Figure 4a and Figure 4b, we take the regular graph in CSR format as the input and condense the columns of each row window (in the red-colored rectangular box) to build TCU blocks (*TC_block*) (*a.k.a.*, the input operand shape of a single MMA instruction), in the orange-colored rectangular box. *nodePointer* is the row pointer array *edgeList* is the edges of each node stored continuously. In this paper, we demonstrate the use of standard MMA shape for TF-32 of TCU on Ampere GPU architecture, and other MMA shapes [41] can also be used under different precision (*e.g.*, `half` and `int8`) and GPU architecture (*e.g.*, Turing).

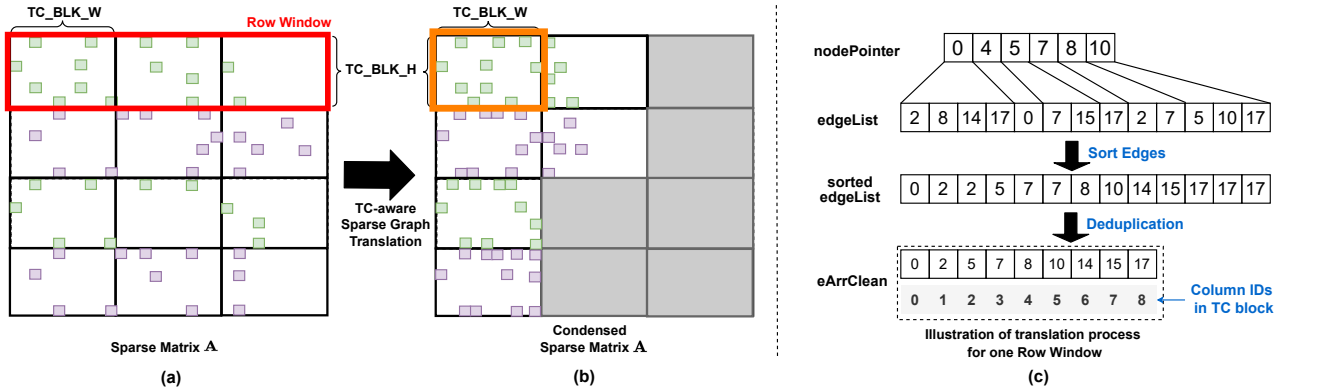


Figure 4: Illustration of Sparse Graph Translation. Note that the grey area indicates the TCU blocks that will be directly skipped.

Algorithm 1: TCU-aware Sparse Graph Translation.

input : Graph adjacent matrix **A** (*nodePointer*, *edgeList*).
output : Result of *winPartition* and *edgeToCol*.

```
/* Compute the total number of row windows.                    */
numRowWin = ceil(numNodes/winSize);
for winId in numRowWin do
    /* EdgeIndex range of the current rowWindow.               */
    winStart = nodePointer[winId * winSize];
    winEnd = nodePointer[(winId + 1) * winSize];
    /* Sort the edges of the current rowWindow.                */
    eArray = Sort(winStart, winEnd, edgeList);
    /* Deduplicate edges of the current rowWindow.             */
    eArrClean = Deduplication(eArray);
    /* #TC blocks in the current rowWindow.                    */
    winPartition[winId] =
      ceil(eArrClean.size/TC_BLK_w);
    /* Edges-to-columnID mapping in TC Blocks.                 */
    for eIndex in [winStart, winEnd] do
        eid = edgeList[eIndex];
        edgeToCol[eIndex] = eArrClean[eid];
    end
end
```

Algorithm 2: TC-GNN Neighbor Aggregation.

input : Condensed graph structure (*nodePointer*, *edgeList*, *edgeToCol*, *winPartition*) and node embedding matrix (**X**).
output : Updated node embedding matrix ($\hat{\mathbf{X}}$).

```
/* Traverse through all row windows.                           */
for winId in numRowWindows do
    /* #TC blocks of the row window.                           */
    numTCblocks = winPartition[winId] ;
    /* Edge range of TC blocks of the row window.              */
    edgeRan = GetEdgeRange(nodePointer, winId);
    for TCblkId in numTCblocks do
        /* The edgeList chunk in current TC block.             */
        edgeChunk = GetChunk(edgeList, edgeRan, TCblkId);
        /* Neighbor node Ids in current TC block.              */
        colToNId = GetNeighbors(edgeChunk, edgeToCol);
        /* Initiate a dense tile (ATile).                      */
        ATile = InitSparse(edgeChunk, winId);
        /* Initiate a dense tile (XTile).                      */
        XTile, colId = FetchDense(colToNId, X);
        /* Compute XnewTile via TCU GEMM.                      */
        XnewTile = TCcompute(ATile, XTile);
        /* Store XnewTile of X̂.                                */
        X̂ = StoreDense(XNewTile, winId, colId);
    end
end
```

SGT takes several steps for processing each row window, as detailed in Algorithm 1 and visualized in Figure 4c. *winPartition* is an array for maintaining the number of TC blocks in each row window. *edgeToCol* is an array for maintaining the mapping between the edges and their corresponding position in the graph after SGT. Note that *edgeToCol* has the same length as *edgeList* but with *column-id* from *eArrClean*. *colToRow* maps *column-id* of adjacency matrices to the *row-id* of embedding matrices. We choose the size of the row window (*winSize=TC_BLK_H*) and column width (*TC_BLK_W*) according to TCU MMA specification (*e.g.*, *TC_BLK_H*=16, *TC_BLK_W*=8 in TF-32). After condensing the graph within each row window, the time complexity of sliding the *TC_block* can be reduced from $O(\frac{N}{TC\_BLK\_W})$ to only $O(\frac{nnz_{unique}}{TC\_BLK\_W})$, where N is the total number of nodes in the graph and nnz_{unique} is the size of the unique neighbor within the current row window, which equals *eArrClean.size* in Algorithm 1. The density (computation intensity) of each identified TCU block can be largely improved. Considering the case in Figure 4, after the sparse graph translation, we can achieve 2× higher density on individual TCU blocks (Figure 4b) compared with the original one (Figure 4a).

Compared to existing sparse matrix formats (e.g., Blocked-Ellpack [37]) which use the regular matrix tiles to cover the irregularly scattered non-zero elements, SGT reduces the irregularity of non-zero-elements layout to fit them into fewer number TCU blocks, thus, reducing the unnecessary computation and memory overhead. SGT is applicable for both the SpMM and SDDMM in GNN sparse operations and can be easily parallelized because the processing of individual row windows is independent. In most cases, SGT only needs to execute once and its result can be reused across many epochs/rounds of GNN training/inference.

Additionally, SGT can be generally used with other ac-

celerators (e.g., AMD-GPUs with matrixCore and TPUs) that offer similar dense MM primitives. CPUs have no direct alternative to TensorCore-like MM primitives. However, with AVX-vectorized instructions, CPUs can benefit from SGT by setting *BLK_H*=1 and *BLK_W*=(#elements-per-AVX-instruction). TC-GNN currently targets GNN training. SGT is conducted once before training. SGT cost can be offset by training iterations (averaged 2% for 200 iterations as DGL).

4.2 TCU-tailored GNN Computation

Besides the effective way to condense the sparse tiles, the next major challenge is *how to tailor the computation schedule of GNN algorithms* so that we can capitalize on the performance of condensed sparse graphs and the powerful TCUs. We focus on two major types of computation in GNNs.

Neighbor Aggregation The major part of GNN sparse computing is neighbor aggregation, which can generally be formalized as SpMM operations by many state-of-the-art frameworks [55]. And they employ the cuSPARSE [38] on CUDA cores as a black-box technique for supporting sparse GNN computation. In contrast, our TC-GNN design targets at TCU for the major neighbor aggregation computation which demands a specialized algorithmic design. TC-GNN focuses on maximizing the net performance gains by gracefully batching the originally highly irregular SpMM as dense GEMM computation and solving it on TCU effectively. As illustrated in Algorithm 2, the node aggregation processes all TC blocks from each row window. *nodePointer* and *edgeList* are directly from graph CSR, while *edgeToCol* and *winPartition* are generated from SGT discussed in the previous section. Note that **InitSparse** is to initialize a sparse tile in dense format according to the translated graph structure of the current TC block. Meanwhile, **FetchDense** returns a dense node embedding matrix tile $XTile$ for TCU computation, and the corresponding column range *colId* (embedding dimension range) of matrix $\mathbf{X}$. This is to handle the case that the width of one $XTile$ could not cover the full-width (all dimensions) of $\mathbf{X}$. Therefore, the *colId* will be used to put the current TCU computation output to the correct location in the updated embedding matrix $\hat{\mathbf{X}}$.

Edge Feature Computing Previous research [51, 54] has demonstrated the great importance of incorporating the edge feature for a better GNN model algorithmic performance (*e.g.*, accuracy, and F1-score). The underlying building block to generate edge features is the Sampled Dense-Dense Matrix Multiplication (SDDMM)-like operation. In TC-GNN, we support SDDMM with the collaboration of the above sparse graph translation and TCU-tailored algorithm design, as described in Algorithm 3. The overall algorithm structure and inputs are similar to the above neighbor aggregation. The major difference is the output. In the case of neighbor aggregation, our output is the updated dense node embedding matrix ($\hat{\mathbf{X}}$), where edge feature computing will generate a sparse output with the same shape as the graph edge lists.

Algorithm 3: TC-GNN Edge Feature Computation.

```
input : Condensed graph data (nodePointer, edgeList, edgeToCol,
        winPartition) and node embedding matrix (X).
output: Edge Feature List (edgeValList).
/* Traverse through all row windows.                          */
for winId in numRowWin do
    /* #TC blocks in the row window.                          */
    numTCblocks = winPartition[winId] ;
    /* Edge range of TC blocks of the row window.             */
    edgeRan = GetEdgeRange(nodePointer, winId);
    for TCblkId in numTCblocks do
        /* EdgeList chunk in current TC block.               */
        edgeChunk = GetChunk(edgeList, edgeRan, TCblkId);
        /* Neighbor node Ids in current TC block.            */
        colToNId = GetNeighbors(edgeChunk, edgeToCol);
        /* Fetch a dense tile (XTile_A).                     */
        XTile_A = FetchDenseRow(winId, TCblkId, X);
        /* Fetch a dense tile (XTile_B).                     */
        XTile_B = FetchDenseCol(colToNId, edgeToCol, X);
        /* Compute edgeValTile via TCU GEMM.                 */
        edgeValTile = TCcompute(XTile_A, XTile_B);
        /* Store edgeValTile to edgeValList.                 */
        StoreSparse(edgeValList, edgeValTile,
                    edgeList, edgeToCol);
    end
end
```

Note that fetching the $XTile_A$ only needs to consecutively access the node embedding matrix $\mathbf{A}$ by rows while fetching the $XTile_B$ requires first computing the TCU block column-id to node-id (*colToNId*) to fetch the corresponding neighbor node embeddings from the same node embedding matrix $\mathbf{X}$.

Despite the dataflow similarity with dense-GEMM computation (e.g., CUTLASS [39]), TC-GNN has to overcome the limited parallelism (imbalance workload) and sparse/irregular access with novel algorithmic and kernel designs. While these challenges are absent in dense-GEMM computation with naturally high parallelism and data-access locality.

4.3 TCU-centric Workload Mapping

In collaborating with our TCU-tailored algorithm design, an effective mapping of our algorithmic design to low-level GPU primitives is indispensable for high-performance delivery. We discuss two key techniques: *GPU-aware Workload Decomposition* and *TCU-optimized dataflow design*.

4.3.1 GPU-aware Workload Decomposition

Different from previous work [13, 55] focusing on CUDA cores only, TC-GNN highlights itself with CUDA core and TCU collaboration through effective two-level workload mapping. The idea is based on the fact that CUDA cores work in SIMT fashion and are operated by individual threads, while TCU designated for GEMM computation requires collaboration from a warp of threads (32 threads). Our key design principle is to *mix these two types of computing units as a single GPU kernel*, which can efficiently coordinate the kernel

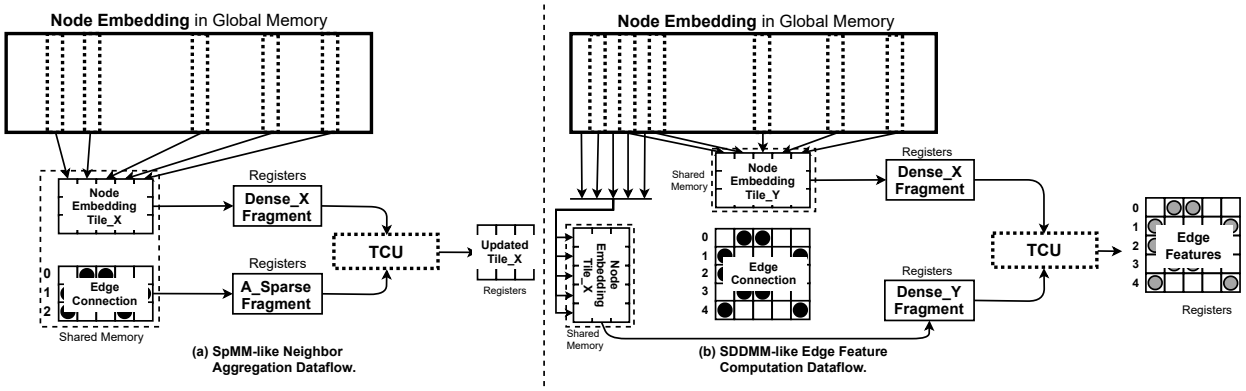


Figure 5: TCU-optimized Dataflow Design for (a) Neighbor Aggregation and (b) Edge Feature Computing in GNNs.

execution at different levels of execution granularity.

In TC-GNN, we operate CUDA cores by thread blocks and manage TCU by thread warps. Specifically, threads running CUDA cores from the same thread block will load data (*e.g.*, edges) from the global memory to shared memory. Note that in our design we assign each row window (discussed in §4.1) to one thread block. The number of threads in each block should be divisible by the number of threads in each warp (32) for better performance. Once threads running on CUDA cores (CUDA-core threads) finish the data loading, threads from each warp (TCU threads) will operate TCU for GEMM computation (including loading the data from the shared memory to thread-local registers (fragments), applying GEMM computation on data in registers, accumulating results on registers, and storing the final results back to global memory). Note that there would be a large overlap of the CUDA-core threads and TCU threads, both of which are threads from the same blocks but running at a different time frames. In general, we use more CUDA-core threads than TCU threads considering that global memory access demands more parallelization.

There are two major benefits of such two-level workload decomposition. First, threads from the same block can work together to improve the memory access parallelization to better utilize memory bandwidth. Second, warps from the same block can reuse the loaded data, including the information (*e.g.*, column index mapping) of the translated graph and the tiles from the dense node embedding matrix. Therefore, we can avoid redundant high-cost global memory operations.

4.3.2 TCU-optimized Dataflow Design

As the major technique to improve the GPU performance, shared memory is customized for our TCU-based sparse kernel design for re-organizing data layout for dense TCU computation and reducing the redundant global memory traffic. Our design takes the TCU specialty into careful consideration from two aspects, 1) the input matrix tile size of the TCU, which is M(16)×N(16)×K(8) in the case of TF-32, and 2) the tile fragment layout for fast computation. The common practice of the loaded tile A and B are stored in row-major and column-major for better performance. Next, we will detail our TCU-optimized dataflow design for both neighbor aggregation and edge feature computation.

Neighbor Aggregation In Figure 5a, shared memory is mainly used for caching several most frequently used information, including the tile of sparse matrix A (`sparse_A`), the column-id of the sparse matrix **A** to row-id of node embedding matrix **X** (`sparse_AToX_index`), and the dense tile of **X** (`dense_X`). When handling each TCU block, we assign all threads from the same block of threads for loading the sparse tile while allowing several warps to concurrently load the dense row tile from the matrix X. The reasons for enforcing such caching are two-fold. First, it can bridge the gap between the sparse graph data and the dense GEMM computing that requires continuous data layout. For example, the adjacent matrix **A** is input as CSR format that **cannot** be directed feed to TCU GEMM computation, therefore, we use a shared memory `sparse_A` to initialize its equivalent dense tile. Similarly, we cache rows of **X** according to the columns of **A** to the row of **X** mapping after our sparse graph translation, where originally scattered columns of **A** (the rows of **X**) are condensed. Second, it can enable data reuse on `sparse_AToX_index` and `sparse_A`. This is because in general, the `BLK_H` (16) cannot cover all dimensions of a node embedding (*e.g.*, 64), multiple warps will be initiated of the same block to operate TCU in parallel to work on non-overlapped dense tiles while sharing the same sparse adjacency matrix tile.

Edge Feature Computation Similar to the shared memory design in neighbor aggregation, for edge feature computing, as visualized in Figure 5b, the shared memory is utilized for `sparse_A`, `sparse_AToX_index`, and `dense_X`. We assign all threads from the same block of threads for loading the sparse tile while allowing several warps to concurrently load the dense row tile from the matrix **X**. Compared with dataflow design in neighbor aggregation, edge feature computing demonstrates several differences.

First, the sizes of `sparse_A` are different. In the neighbor aggregation computation, the sparse matrix **A** is used as

Table 4: Datasets for evaluation.

Type	Dataset	Abbr.	#Vertex	#Edge	Dim.	#Class
I	Citeseer	CR	3,327	9,464	3703	6
	Cora	CO	2,708	10,858	1433	7
	Pubmed	PB	19,717	88,676	500	3
	PPI	PI	56,944	818,716	50	121
II	PROTEINS_full	PR	43,471	162,088	29	2
	OVCAR-8H	OV	1,890,931	3,946,402	66	2
	Yeast	YT	1,714,644	3,636,546	74	2
	DD	DD	334,925	1,686,092	89	2
	YeastH	YH	3,139,988	6,487,230	75	2
III	amazon0505	AZ	410,236	4,878,875	96	22
	artist	AT	50,515	1,638,396	100	12
	com-amazon	CA	334,863	1,851,744	96	22
	soc-BlogCatalog	SC	88,784	2,093,195	128	39
	amazon0601	AO	403,394	3,387,388	96	22

one operand in the SpMM-like computation, therefore, the minimal processing granularity is 16×8, while in edge feature computing by following SDDMM-like operation, the sparse matrix $\mathbf{A}$ serves as the output matrix, thus, maintaining the minimum processing granularity is 16×16. To reuse the same translated sparse graph as SpMM, we need to recalculate the total number of TC blocks. ***Second***, iterations along the embedding dimension would be different. Compared with neighbor aggregation, edge feature computing requires the result accumulation along the embedding dimension. The result will only be output until all iterations have finished. In neighbor aggregation, the node embedding vector is divided among several warps, each of which will output their aggregation result to non-overlapped embedding dimension ranges in parallel. ***Third***, the output format has changed. Compared with SpMM-like neighbor aggregation which directly output computing results as an updated dense matrix $\hat{\mathbf{X}}$, SDDMM-like edge feature computing requires a sparse format (the same shape as `edgeList`) output for compatibility with neighbor aggregation and memory space. Therefore, one more step of dense-to-sparse translation is employed.

5 Evaluation

Benchmarks: We choose two representative GNN models widely used by previous work [13, 34, 55] on *node classification* tasks. Specifically, 1) Graph Convolutional Network (**GCN**) [27] is one of the most popular GNN model architectures. It is also the key backbone for many other GNNs (e.g., GraphSAGE [19] and differentiable pooling (Diffpool) [61]). Therefore, improving the performance of GCN will also benefit a broad range of GNNs. For GCN evaluation, we use the setting: *2 layers with 16 hidden dimensions per layer*, which is also the setting from the original paper [27]. 2) Attention-based Graph Neural Network (**AGNN**) [51]. AGNN differs from GCN in its aggregation function, which computes edge features (via embedding vector dot-product between source and destination vertices) before the node aggregation. AGNN is also the reference architecture for many other recent GNNs for better model algorithmic performance. For AGNN, we use: *4 layers with 32 hidden dimensions per layer*.

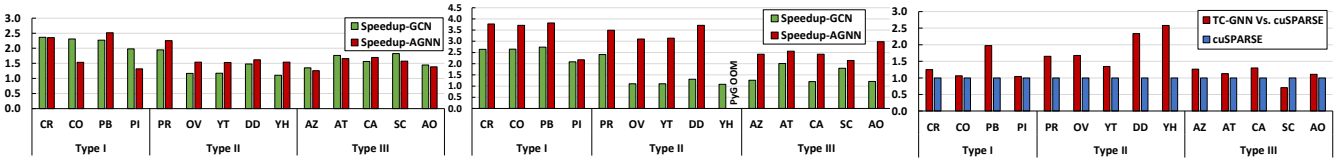
Baselines: 1) Deep Graph Library (**DGL**) [55] is the state-of-the-art GNN framework on GPUs, which is built with the high-performance CUDA-core-based cuSPARSE [38] library as the backend and uses PyTorch [49] as its front-end programming interface. DGL significantly outperforms other existing GNN frameworks [13] over various datasets on many mainstream GNN model architectures. Therefore, we make an in-depth comparison with DGL. 2) PyTorch Geometric (**PyG**) [13] is another GNN framework. PyG leverages torch-scatter [14] library (highly-engineered CUDA-core kernel) as the backend support, which highlights its performance on batched small graph settings; 3) Blocked-SpMM [37] (**bSpMM**) accelerates SpMM on TCU. It is included in the recent update on the cuSPARSE library. bSpMM requires the sparse matrix with Blocked-Ellpack format for computation. Its computation on non-zero blocks can be seen as the hybrid sparse-dense solution (§3.3). Note that the bSpMM has not been incorporated into any existing GNN frameworks. We also compare TC-GNN with **tSparse** [62] and **Triton** [52] for non-vendor-developed highly optimized kernels on TCUs.

Datasets, Platforms, and Metrics: We cover three types of datasets (Table 4), which have been used in previous GNN-related work [13, 34, 55]. Specifically, **Type I** graphs are the typical datasets used by previous GNN algorithm papers [19, 27, 59]. They are usually small in the number of nodes and edges, but rich in node embedding information with high dimensionality. **Type II** graphs [26] are the popular benchmark datasets for graph kernels and are selected as the built-in datasets for PyG [13]. Each dataset consists of a set of small graphs, which only have intra-graph edge connections without inter-graph edge connections. **Type III** graphs [27, 30] are large in terms of the number of nodes and edges. These graphs demonstrate high irregularity in its structures, which are challenging for most of the existing GNN frameworks. The core design of TC-GNN consists of around 2.5K lines of code. TC-GNN backend is implemented with C++ and CUDA C, and its front end is implemented in Python. Our major evaluation platform is a server with an 8-core 16-thread Intel Xeon Silver 4110 CPU and an NVIDIA RTX3090 GPU. To measure the performance speedup, we calculate the average latency of 200 end-to-end runs.

5.1 Compared with DGL

Figure 6a shows that TC-GNN achieves 1.70× speedup on average compared to DGL over three types of datasets across GCN and AGNN models on end-to-end training. Our kernel profiling via Nsight Compute shows that TC-GNN achieves high SM occupancy (averaged 85.28%), which is on average 21.05% higher compared to DGL across all datasets.

Type I Graphs: The performance improvements against DGL are significantly higher for GCN (on average 2.23×) compared to AGNN (on average 1.93×). The major reason



(a) (b) (c)

Figure 6: Speedup over (a) DGL and (b) PyG on GCN and AGNN; (c) Speedup over cuSPARSE bSpMM on TCUs.

is their different GNN computation patterns. GCN only consists of a neighbor aggregation phase (SpMM-like operation) and a node update phase (GEMM operation). Whereas in the AGNN, the aggregation phase would also require an additional edge attention value (feature) computation based on SDDMM-like operations. Compared with SpMM-like operations, edge attention computation in SDDMM is more sensitive to the irregular sparse graph structure because of much more intensive computations and memory access. Thus, the performance improvement is relatively lower.

Type II Graphs: TC-GNN achieves averaged 1.38× speedup on GCN and 1.70× speedup on AGNN for the Type II graphs. Speedup on Type II graphs is relatively lower compared with Type I, since Type II datasets consist of a set of small graphs with very dense intra-graph connections but no inter-graph edges. This leads to a lower benefit from the sparse graph translation that would show more effectiveness on highly irregular and sparse graphs. Such a clustered graph structure would also benefit cuSPARSE due to more efficient memory access, *i.e.*, less irregular data fetching from the sparse matrix. In addition, for AGNN, TC-GNN can still demonstrate evident performance benefits over the DGL (CUDA core only) that can mainly contribute to TCU-based SDDMM-like designs that can fully exploit the power of GPU through an effective TCU and CUDA core collaboration.

Type III Graphs: The speedup is also evident (on average 1.59× for GCN and average 1.51× for AGNN) on graphs with a large number of nodes and edges and irregular graph structures. The reason is the high overhead global memory access can be well reduced through our spare graph translation. Besides, our dimension-split strategy further facilitates efficient workload sharing among warps by improving the data spatial/temporal locality. On the dataset AT and SC, which have a higher average degree within Type III datasets, we notice a better speedup performance for both GCN and AGNN. This is because 1) more neighbors per node can lead to a higher density of non-zero elements within each tile/fragment. Thus, it can fully exploit the computation benefits of each TCU GEMM operation; 2) it can also facilitate more efficient memory access. For example, in AGNN, fetching one dense embedding x from the dense matrix X can be reused more times by applying a dot-product between x and many columns of the dense matrix X^T (neighbors embeddings).

Table 5: Compare TC-GNN with tSparse and Triton.

Dataset	tSparse (ms)	Triton (ms)	TC-GNN (ms)
AZ	18.60	31.64	4.09
AT	9.15	12.86	3.06
CA	13.84	15.50	3.26
SC	9.74	14.38	3.59
AO	11.93	21.78	3.41

Additionally, our performance breakdown analysis shows that for graphs with highly scattered and irregular edge distribution, such as Type I and III graphs, SGT would contribute more (averaged 64%) to the overall performance improvements since it helps significantly reduce the unnecessary workload. For graphs with highly dense and more regular edge connections, such as Type II datasets, SGT contributes relatively minor (averaged 23%) to the overall performance since it could not squeeze out more redundant computations from already condensed edge tiles.

5.2 Compared with other baselines

Compared with PyG Figure 6b shows TC-GNN can outperform PyG with an average of 1.76× speedup on GCN and an average of 2.82× speedup on AGNN. For GCN, TC-GNN achieves significant speedup on datasets with high-dimensional node embedding, such as *Yeast (YT)*, through effective TCU acceleration through a TCU-aware sparse graph translation while reducing the synchronization overhead by employing our highly parallelized TCU-tailored algorithm design. PyG, however, achieves inferior performance because its underlying GPU kernel can only leverage CUDA cores, thus, intrinsically bounded by CUDA core performance.

Compared with cuSPARSE bSpMM Figure 6c shows that TC-GNN outperforms bSpMM with on average 1.76× speedup on neighbor aggregation and improves effective computation by 75.8% on average. Our SGT technique can maximize the non-zero density of each non-zero tile and significantly reduce the number of non-zero tiles to be processed. However, bSpMM in cuSPARSE has to comply with the strict input sparse pattern (indicated in official API documentation [42]). For example, all rows in the arrays must have the same number of non-zero blocks. Thus, more redundant computations (on padding non-structural non-zero blocks) in bSpMM lead to inferior performance. We also notice that for

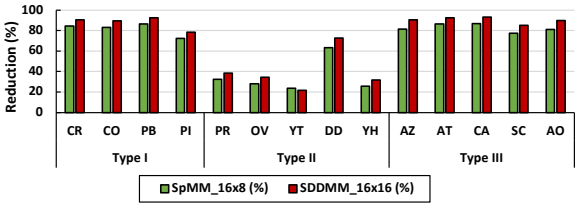


Figure 7: SGT Effectiveness on SpMM and SDDMM.

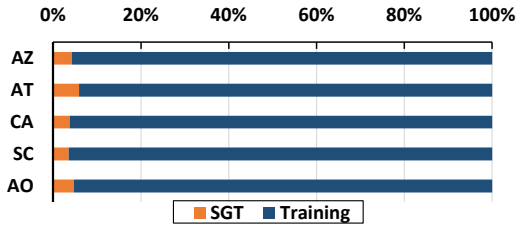


Figure 8: The overhead analysis of SGT.

SC datasets with a high average node degree and clustered node distribution, bSpMM would benefit more due to its usage of a larger block size of 32×32 (fewer TCU invocations) compared to 16×8 in TC-GNN (more TCU invocations).

Compared with tSparse and Triton From Table 5, TC-GNN can outperform tSparse with on average 3.60× speedup on SpMM. The major reason behind this is that TC-GNN can well reduce the graph structural-level irregularity through our novel SGT strategy to benefit the dense TCU-based computation. In contrast, tSparse only considers partitioning the input sparse matrix into dense/sparse tiles based on their non-zero elements but ignores the potential of compressing non-zero elements into fewer tiles to reduce the workload. TC-GNN also outperforms Triton with on average 5.42× speedup on SpMM. Triton's block-sparse GEMM for TCU acceleration is designed for dense neural networks (focusing on feature maps' sparsity), which is quite different from GNNs (focusing on the graph adjacency matrix's sparsity) with significantly larger sparse matrix size and more irregular pattern.

5.3 Additional Studies

SGT Effectiveness & Overhead We conduct a quantitive analysis of SGT in terms of the total number of TCU blocks between graphs w/o SGT and the graphs w/ SGT applied. Note that in the SpMM-based aggregation, the size of TCU blocks is 16×8 since it serves as one of the operands in TCU GEMM. While in SDDMM-based edge feature computation, the size of TCU blocks is 16×16 since it serves as the resulting matrix of TCU GEMM. Figure 7 shows that across all types of datasets, our SGT technique can significantly reduce the number of traversed TCU blocks (on average 67.47%). The major reason is that SGT can largely improve the density of non-zero elements within each TCU block. In contrast, the graphs w/o SGT would demonstrate a large number of highly sparse TCU blocks. What is also worth noticing is that on Type II graphs, such a reduction benefit is lower. The reason is that Type II graphs consist of a set of small subgraphs that only maintain the intra-subgraph connections, which already maintain dense columns. We evaluate the overhead of SGT (Figure 8), we find that its overhead is consistently low (on average 4.43%) compared with the overall training time (200 epoches as DGL [55]).

Sparsity Analysis We compare with bSpMM on synthetic matrix data with different sparsity (zero-element ratio). Note that we change the sparsity by varying the number of dense non-zero blocks (16×16) within each row window, the input adjacent matrix size is fixed to 4096×4096 while the dense embedding matrix dimension is fixed to 16. Table 6 shows that when sparsity increases from 93.75% to 99.61%, TC-GNN design demonstrates more throughput performance strength (averaged 6.9×) and this is also the common sparsity range (more than 95%) for most input graphs of GNNs. When the sparsity drops to around 87.50% the sparse would demonstrate more advantage due to more dense blocks for computation.

Table 6: Sparsity Analysis. Numbers for bSpMM/TC-GNN are in GFLOPs. "DB/W": dense blocks per row window.

DB/W	Sparsity (%)	bSpMM	TC-GNN
1	99.61	773.86	12,686.02
2	99.22	1,597.83	11,010.75
4	98.44	3,348.75	18,164.08
8	96.88	6,528.10	25,883.10
16	93.75	12,955.40	23,865.99
32	87.50	26,061.70	16,629.28

Warps per Block: Figure 9 shows that with the increase of the number of warps, the overall performance for training per epoch would first decrease due to the better parallelism for loading the graph data. However, the number of warps per block would decrease the overall performance under certain circumstances (e.g., 32). All three settings suffer from evident performance degradation. Because the global memory access contention will become severe, leading to lower execution performance. Different datasets would have different "optimal" choices of the warp-per-block parameter. For example, on the CA dataset, 2 warps per block can deliver the best performance, while AZ requires 8 warps per block. Based on our profiling and empirical study, the selection of this parameter should consider the average *#edges* per row window (*avg.edges*), which can be easily get during the preprocessing. Our `preprocessor` will generate $warpPerBlock = \lfloor \frac{avg.edge}{32} \rfloor$ to approach the "optimal" performance. For instance, the average edges per row window are 88 for CA, it reaches the best performance at 2 warps per block.

Throughput Analysis: For sparse matrix computations in GNNs, we measure the throughput performance of SpMM in TC-GNN when the dimension of node embedding increases for a roofline analysis. Because sparse matrix computation is largely limited by its memory access performance, which

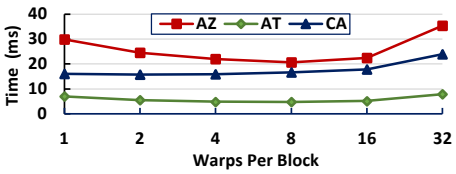


Figure 9: Performance Impact of Warps per Block.

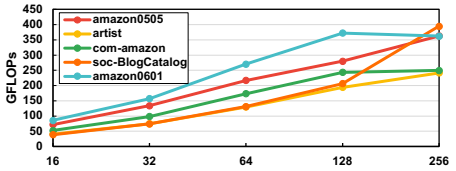


Figure 10: Analysis of TC-GNN kernel throughput when increasing the node embedding dimension from 16 to 256.

is quite different from the dense GEMM computation that is largely bounded by the computing performance. Figure 10 shows that the throughput of TC-GNN can scale proportionally with the growing number of node embedding dimensions. This also indicates that TC-GNN can effectively handle the graphs with high-dimensional node embeddings and well utilize GPU resources.

6 Related Work and Discussion

Other GPUs TC-GNN can easily generalize to other GPUs (e.g., A6000, H100, and RTX4090) with TCUs via recompilation (`python setup.py install`). TC-GNN also supports different TCU configurations (e.g., precision) by modifying (*BLK_H*, *BLK_W* in `TCGNN_conv/config.h`) and four parameters (`M, N, K, dataType`) in `wmma::fragment`, then recompile. For future GPUs with more TCUs, our TC-GNN can also be adapted to accommodate such changes and maintain its performance advantage. There are two future GPU designs that we anticipate. The first direction is to place more TCUs per SM while keeping the total number of SMs unchanged. There will be more active warps per thread block (This is mainly because TCUs are operated by warps) and each warp will process fewer neighbors. The cost of decomposition and mapping can be offset by parallelism among more warps. The second direction is to place more SMs on GPUs while keeping TCUs per GPU unchanged. In this scenario, there will be more thread blocks and each thread block will process neighbors from fewer nodes. The cost can be offset by parallelism among more thread blocks.

Other GNN Frameworks Besides DGL and PyG, other single-GPU GNN frameworks like GNNAdvisor [57], GE-SpMM [21], and fuseGNN [7], tailor their own GNN layers manually with low-level GPU kernel optimizations. Unfortunately, these designs limit their kernel optimizations to CUDA cores, thus, missing the golden opportunities to exploit the full potential of widely deployed AI-tailored GPUs with TCUs.

Graph Partitioning/Reordering ROC [23] introduces a learning-based graph partitioning to reduce the data movement between CPU and GPU when processing large graphs. Rabbit Order [4] and Reverse Cuthill Mckee Algorithm [9] are focusing on *row reordering/clustering* to improve node/row-wise computation locality. Our sparse-graph translation (SGT) technique is orthogonal and complementary to these graph partitioning and reordering techniques since our SGT focuses on *column (neighbor) re-indexing* to improve neighbor-wise locality for TCU computation.

Distributed GNN Computation There are two major ways of scaling-up GNN computing: 1) *Distributed sampled graphs* [13, 35, 55, 60] (where graph nodes and their embeddings are on the same GPU): TC-GNN can be incorporated directly since all sampled graphs along with their node embeddings are presented at the same GPU. 2) *Distributed full-graph* [15, 23, 34, 58] (where graph nodes and their embeddings may be on different GPUs): TC-GNN needs to be modified slightly by incorporating inter-GPU communication techniques (e.g., Unified Virtual Memory [46] and NVSHMEM [44]) to support the remote neighbor embedding access. We leave such exploration for our future work.

7 Conclusion

In this paper, we introduce TC-GNN, the first GNN acceleration framework on TCU of GPUs. We design a novel sparse graph translation technique to gracefully fit the sparse GNN workload on dense TCUs. Our TCU-tailored GPU kernel design maximizes the TCU performance gains for GNN computing through effective CUDA core and TCU collaboration and a set of memory/data flow optimizations. Our seamless integration with the PyTorch framework further facilitates end-to-end GNN computing with high programmability. Extensive experiments demonstrate the performance advantage of TC-GNN over the state-of-the-art frameworks. across diverse GNN models and datasets.

Furthermore, our TC-GNN design could also inspire potential TCU-like hardware features that can support (i) the dynamic shape of TCU input tiles and (ii) the dynamic structural sparsity of input tiles to yield higher performance benefits at the runtime. These proposed hardware features will help reduce more unnecessary computation in a more fine-grained and precise manner.

8 Acknowledgment

We would like to thank our shepherd, Asim Kadav, and the anonymous USENIX ATC reviewers. This work was supported in part by NSF-2124039 and CloudBank [36]. We also appreciate the generous help and support from NVIDIA Graduate Fellowship 2022-2023 for Yuke Wang and Amazon Faculty Award 2021-2022 for Yufei Ding.

References

[1] *Intel Math Kernel Library. Reference Manual*. Intel Corporation. Santa Clara, USA.

[2] AMD. All-new matrix core technology for hpc and ai. https://amd.com/en/technologies/cdna.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1999.

[4] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[5] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 1997. Computational algebra and number theory (London, 1993).

[6] Hsinchun Chen, Xin Li, and Zan Huang. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)*, 2005.

[7] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. fusegnn: accelerating graph convolutional neural network training on gpgpu. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020.

[8] De Cheng, Yihong Gong, Xiaojun Chang, Weiwei Shi, Alexander Hauptmann, and Nanning Zheng. Deep feature learning via structured graph laplacian embedding for person re-identification. *Pattern Recognition*, 2018.

[9] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, 1969.

[10] 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.

[11] Alberto Garcia Duran and Mathias Niepert. Learning graph representations with embedding propagation. In *Advances in neural information processing systems (NeurIPS)*, 2017.

[12] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. Egemm-tc: Accelerating scientific computing tensor cores with extended precision. *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, 2021.

[13] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds (ICLR)*, 2019.

[14] Matthias Fey and Jan E. Lenssen. Pytorch extension library of optimized scatter operations, 2019.

[15] Swapnil Gandhi, Anand Padmanabha Iyer, Henry Xu, Theodoros Rekatsinas, Shivaram Venkataraman, Yuan Xie, Yufei Ding, Keval Vora, Ravi Netravali, Miryung Kim, et al. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[16] Jaume Gibert, Ernest Valveny, and Horst Bunke. Graph embedding in vector spaces by node attribute statistics. *Pattern Recognition*, 2012.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[18] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM international conference on Knowledge discovery and data mining (SIGKDD)*, 2016.

[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems (NeurIPS)*, 2017.

[20] 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.

[21] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[22] Zexi Huang, Arlei Silva, and Ambuj Singh. A broader picture of random-walk based graph embedding. In *Proceedings of the 27th ACM International Conference on Knowledge Discovery & Data Mining (SIGKDD)*, 2021.

[23] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and

performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

[24] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture (ISCA)*, 2017.

[25] Riesen Kaspar and Bunke Horst. *Graph classification and clustering based on vector space embedding*. World Scientific, 2010.

[26] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016.

[27] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2017.

[28] Jérôme Kunegis and Andreas Lommatzsch. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009.

[29] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and P. Sadayyapan. Efficient tiled sparse matrix multiplication through matrix signatures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[30] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[31] Ang Li and Simon Su. Accelerating binarized neural networks via bit-tensor-cores in turing gpus. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2020.

[32] Dijun Luo, Chris Ding, Heng Huang, and Tao Li. Non-negative laplacian embedding. In *2009 Ninth IEEE International Conference on Data Mining (ICDM)*, 2009.

[33] Dijun Luo, Feiping Nie, Heng Huang, and Chris H Ding. Cauchy graph embedding. In *Proceedings of the 28th International Conference on Machine Learning*, 2011.

[34] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[35] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses. *arXiv preprint arXiv:2101.07956*, 2021.

[36] Michael Norman, Vince Kellen, Shava Smallen, Brian DeMeulle, Shawn Strande, Ed Lazowska, Naomi Alterman, Rob Fatland, Sarah Stone, Amanda Tan, et al. Cloudbank: Managed services to simplify cloud access for computer science research and education. In *Practice and Experience in Advanced Research Computing*. 2021.

[37] Nvidia. Accelerating matrix multiplication with block sparse format and nvidia tensor cores. https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with\-block-sparse-format-and-nvidia-tensor-cores/.

[38] Nvidia. Cuda sparse matrix library (cusparse). https://developer.nvidia.com/cusparse.

[39] NVIDIA. Cuda template library for dense linear algebra at all levels and scales (cutlass).

[40] Nvidia. Dense linear algebra on gpus. https://developer.nvidia.com/cublas.

[41] NVIDIA. Improved tensor core operations. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#tensor-operations.

[42] Nvidia. Nvidia blocked-sparse api. https://docs.nvidia.com/cuda/cusparse/index.html#cusparse-generic-function-spmm).

[43] Nvidia. Nvidia volta. https://en.wikipedia.org/wiki/Volta_(microarchitecture).

[44] Nvidia. Nvshmem communication library. https://developer.nvidia.com/nvshmem.

[45] NVIDIA. Tensorfloat-32 in the a100 gpu accelerates ai training, hpc up to 20x.

[46] NVIDIA. Unified memory for cuda. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[47] Nvidia. Warp matrix multiply-accumulate (wmma). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma.

[48] NVIDIA. Programming tensor cores in cuda 9. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/, 2017.

[49] 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 *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[50] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2014.

[51] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. Attention-based graph neural network for semi-supervised learning. 2018.

[52] Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

[53] Tomasz Tylenda, Ralitsa Angelova, and Srikanta Bedathur. Towards time-aware link prediction in evolving social networks. In *Proceedings of the 3rd workshop on social network mining and analysis*, 2009.

[54] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[55] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[56] Yuke Wang, Boyuan Feng, and Yufei Ding. Qgtc: accelerating quantized graph neural networks via gpu tensor core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2022.

[57] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[58] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Ang Li, Kevin Barker, and Yufei Ding. Mgg: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-gpu platforms. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2023.

[59] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

[60] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[61] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[62] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. Accelerating sparse matrix–matrix multiplication with gpu tensor cores. *Computers & Electrical Engineering*, 2020.

[63] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems (NeurIPS)*, 31, 2018.

A Artifact Appendix

TC-GNN is the first TCU-based GNN acceleration design on GPUs. ***At the input level***, TC-GNN is equipped with a new *sparse graph translation* (SGT) technique that can effectively identify those non-zero tiles and condense non-zero elements from these tiles into fewer number of "dense" tiles. ***At the GPU kernel level***, TC-GNN exploits the benefits of CUDA core and TCU collaboration. The major design idea is that the CUDA core, which is more powerful at fine-grained thread-level execution, would be a good candidate for managing memory-intensive data access. While TCU, which is more powerful in handling simple arithmetic operations (*e.g.*, multiplication and addition), would be well-suited for compute-intensive GEMM on dense tiles generated from SGT. ***At the framework level***, TC-GNN is integrated with the popular PyTorch framework to reduce extra learning efforts and improve user productivity and code portability.

- Code repository: **Github**[2] and **Zenodo**[3].
- **Hardware, OS & Compiler**:
  - Intel Xeon Sliver 4110 CPU (8-core 16-threads) with 64GB host memory, NVIDIA RTX3090 GPU with 24 GB device memory.
  - Operating systems and versions: Ubuntu 16.04+.
  - Compilers and versions: NVCC-11.1+, GCC-7.5.0+ Libraries and versions: CUDA-11.1+, Pytorch-1.8.0, DGL-v0.6.0, PyG-1.6.3 Input datasets and versions: SNAP network datasets.

Step-1: Environment Setup

- 1.1a. [Method-1] Install via Docker (Recommended).

```
cd docker/
./launch.sh
```

- 1.1b. [Method-2] Install via Conda.

```
curl -O https://repo.anaconda.com/archive/Anaconda3
     -2021.05-Linux-x86_64.sh
bash Anaconda3-2019.03-Linux-x86_64.sh
source ~/.bashrc
conda create -n env_name python=3.6
conda install pytorch torchvision torchaudio cudatoolkit
     =11.1 -c pytorch -c conda-forge
conda install -c dglteam dgl-cuda11.0
pip install torch requests tqdm
pip install torch-scatter -f https://pytorch-geometric.
     com/whl/torch-1.8.0+cu111.html
pip install torch-sparse -f https://pytorch-geometric.
     com/whl/torch-1.8.0+cu111.html
pip install torch-cluster -f https://pytorch-geometric.
     com/whl/torch-1.8.0+cu111.html
pip install torch-spline-conv -f https://pytorch-
     geometric.com/whl/torch-1.8.0+cu111.html
pip install torch-geometric
```

[2] https://github.com/YukeWang96/TC-GNN_ATC23.git
[3] https://doi.org/10.5281/zenodo.7893174

- 1.2. Install TC-GNN.

```
cd TCGNN_conv/
./0_build_tcgnn.sh
```

- 1.3. Download Datasets.

```
wget https://storage.googleapis.com/graph_dataset/tcgnn-
     ae-graphs.tar.gz
tar -zxvf tcgnn-ae-graphs.tar.gz
rm -rf tcgnn-ae-graphs.tar.gz
```

Step-2. Run Major Experiments.

- 2.1. TC-GNN model End-to-End.

```
./0_run_tcgnn_model.sh
```

Results: `1_bench_gcn.csv` and `1_bench_agnn.csv`.

- 2.2. DGL baseline (Fig-6a).

```
cd dgl_baseline/
./0_run_dgl.sh
```

Results: `Fig_6a_dgl_gcn.csv` and `Fig_6a_dgl_agnn.csv`.

- 2.3. TC-GNN single kernel.

```
./0_run_tcgnn_single_kernel.sh
```

Results: `1_bench_gcn.csv` and `1_bench_agnn.csv`.

- 2.4. cuSPARSE-bSpMM Baseline (Fig-6c).

```
cd TCGNN-bSpmm/cusparse
./0_run_bSpMM.sh
```

Results: `Fig_6c_cuSPARSE_bSpMM.csv`.

- 2.5. Dense Tile Reduction (Fig-7).

```
python 3_cnt_TC_blk_SDDMM.py
python 3_cnt_TC_blk_SpMM.py
```

Results: `3_cnt_TC_blk_SDDMM.csv` and `3_cnt_TC_blk_SDDMM.csv`.

- 2.6. tSparse Baseline (Table-5, column-2).

```
cd TCGNN-tsparse/
./0_run_tSparse.sh
```

Result: `Table_5_tSparse.csv`.

- 2.7. Triton Baseline (Table-5, column-3).

```
cd TCGNN-trition/python/bench
./0_run_triton.sh
```

Result: `1_run_triton.csv`.

Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training

Jie Sun[1], Li Su[2], Zuocheng Shi[1], Wenting Shen[2], Zeke Wang[1]
Lei Wang[2], Jie Zhang[1], Yong Li[2], Wenyuan Yu[2], Jingren Zhou[2], Fei Wu[1,3]

[1] *Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China*

[2] *Alibaba Group*

[3] *Shanghai Institute for Advanced Study of Zhejiang University, China*

Abstract

Graph neural network(GNN) has been widely applied in real-world applications, such as product recommendation in e-commerce platforms and risk control in financial management systems. Several cache-based GNN systems have been built to accelerate GNN training in a single machine with multiple GPUs. However, these systems fail to train billion-scale graphs efficiently, which is a common challenge in the industry. In this work, we propose Legion, a system that automatically pushes the envelope of multi-GPU systems for accelerating billion-scale GNN training. First, we design a hierarchical graph partitioning mechanism that significantly improves the multi-GPU cache performance. Second, we build a unified multi-GPU cache that helps to minimize the PCIe traffic incurred by caching both graph topology and features with the highest hotness. Third, we develop an automatic cache management mechanism that adapts the multi-GPU cache plan according to the hardware specifications and various graphs to maximize the overall training throughput. Evaluations on various GNN models and multiple datasets show that Legion supports training billion-scale GNNs in a single machine and significantly outperforms the state-of-the-art cache-based systems on small graphs.

1 Introduction

Graph neural networks (GNNs), such as [8, 10, 16, 22, 40, 50], are a class of deep learning algorithms that learn the low-dimensional embedding using the structure and attribute information of graphs. The learned embedding can be further used in machine-learning tasks including node classification and link prediction. GNNs have been successfully applied in many real-world applications, such as recommendation systems in e-commerce platforms, fraud detection and risk control in financial management, and molecular property prediction in drug development [13, 25, 37, 48, 49]. Systems such as DGL [42], PyG [31], and Graph-Learn [55] are proposed to ease the development and training of GNN models.

It is common to apply GNNs over large-scale graphs in industrial scenarios. For example, in Alibaba's Taobao recommendation system, the user behavior graph contains more than one billion vertices and tens of billions of edges [55]. In addition, as graphs are often skewed, it is infeasible to aggregate all neighboring vertices when training a specific vertex. Sampling-based mini-batch training, such as GraphSAGE [16], is proposed to extend GNN training to very large graphs. In the sampling-based GNN training, there are two key steps of data preparations before training a batch: (1) sampling the multi-hop sub-graph for each vertex in the batch, and (2) extracting the features of vertices in sampled sub-graphs. Systems such as DGL [42] and PyG [31] store the graph data in the CPU memory, prepare the training data of mini-batches using CPUs, and utilize GPUs for model training. As this approach requires transferring the sampled sub-graphs and high-dimension feature data to the GPU for every batch, the end-to-end training throughput is severely limited by the CPU-GPU data transferring bandwidth [23, 47]. In addition, the throughput of graph sampling using CPU is often insufficient to keep up with the throughput of GPU training, especially in multi-GPU machines.

Several cache-based approaches have been proposed to speed up GNN training [23, 29, 33, 47]. As it is the feature data that accounts for a majority of the CPU-GPU data transferring, caching the features of frequently accessed vertices in GPU can significantly reduce the amount of transferred data. To improve the throughput of graph sampling, GPU-based sampling has also been adopted in GNN systems [33, 42, 47].

We identify that existing approaches face severe limitations or performance issues in multi-GPU training, particularly when the graph is large. First, the multi-GPU cache scalability of existing cache-based systems is poor. Some cache-based GNN systems [33, 47] shuffle the training set across all GPUs and replicate an identical feature cache across all GPUs or NVLink cliques[1] to facilitate data parallel training. The cache capacity is constrained by the memory of a single GPU or

[1] NVLink clique denotes a group of GPUs where each pair of GPUs are connected with NVLink.

NVLink clique (an NVLink clique only consists of two GPUs in some multi-GPU architectures), resulting in poor cache performance when scaling up the number of GPUs (see the experiment in Figure 2). PaGraph [23] partitions the graph using a self-reliant algorithm and caches nodes with the highest in-degree for different partitions in different GPUs, trying to make use of data locality inside each partition. As partitions in PaGraph include the complete L-hop neighbors of their training vertices, there is a significant overlap between the caches of different partitions, resulting in the same duplication issue as the aforementioned cache-based GNN systems. Second, when adopting GPU-based graph sampling, existing systems manage the graph topology in a very coarse-grained manner: the topology has to be completely stored in a single GPU [33, 42, 47] or in the CPU memory [33, 42]. The former approach puts a hard limit on the graph scale, and further squeezes the cache capacity for features. The latter storing the topology in the CPU and accessing it from GPU would result in very low utilization of the PCIe bandwidth, as the data access of graph sampling is usually random and fine-grained.

This paper presents Legion, a GNN system that fully explores the hardware capabilities of modern multi-GPU servers for training large-scale graphs in a single machine. Legion proposes two key designs to fully exploit the memory space of multi-GPUs for feature and topology cache. First, to avoid cache replication, we propose **NVLink-aware hierarchical graph partitioning** technique that helps scale the cache on multi-GPU memory efficiently according to the specific hardware structure. Legion first partitions the graph with minimal edge-cut and assigns each partition exclusively to an NVLink clique, and then uses hash partition to further map the training vertices to GPUs inside each NVLink clique. Second, we propose a **hotness-aware unified cache** that manages both the feature and topology cache in a vertex-centric data structure. We enable an NVLink-enhanced cache space for the unified cache and prioritize the topology and features with the highest hotness to fill the cache, so as to improve the multi-GPU memory utilization.

The above designs pose a new challenge to Legion. Given a fixed size of GPU memory, it is hard to manually decide the optimal fractions of topology and feature cache such that the overall training throughput is maximized. To solve the challenge, we propose an **automatic cache management** mechanism. Specifically, we build a cost model in the mechanism to evaluate the key factor to the overall throughput, i.e., PCIe traffic, of both graph sampling and feature extraction in the training phase, which is used to guide the allocations of cache spaces for graph topology and feature. Overall, the three key designs in Legion enable automatic caching optimization and full utilization of hardware capability of various modern GPU servers. Experiments show that Legion can outperform state-of-the-art cache-based GNN systems up to 4.32×.

In summary, the contributions of this paper include:

1. We propose an NVLink-aware hierarchical graph partitioning technique that helps minimize cache replication between NVLink cliques and extends the threshold of cache capacity beyond the limit of an NVLink clique.
2. We propose a hotness-aware unified cache to store topology and features with the highest hotness in GPU memory, so as to improve the GPU memory utilization.
3. We present an automatic cache management mechanism that searches for the optimal cache plan without requiring extra knowledge of hardware specifications and GNN performance details from users.
4. We implement Legion that fully explores the hardware capabilities of multi-GPU systems targeting billion-scale GNN training, not supported by existing cache-based GNN systems, in a single server.

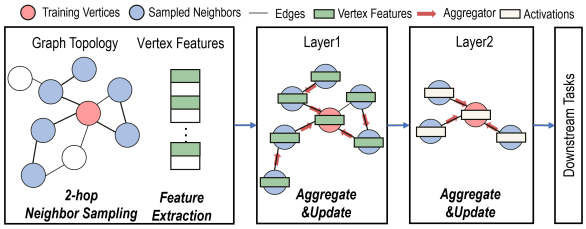


Figure 1: The workflow of 2-hop GraphSAGE training.

2 Preliminaries

In this section, we introduce the basic concept of GNN and the workflow of mini-batch GNN training.

2.1 Graph Neural Networks

Given a graph $G = (V, E)$, where each vertex is associated with a vector of data as its features $X_v, v \in V$, Graph Neural Networks(GNNs) learn a low-dimensional embedding for each target vertex by stacking multiple GNNs layers L. For each layer $l, l \in L$, vertex v updates its activation by aggregating features or hidden activations of its neighbors $N(v), v \in V$:

$$\begin{aligned} a_v^l &= AGGREGATE^l(h_u^{l-1} | u \in N(v)) \\ h_v^l &= UPDATE^l(a_v^l, h_v^{l-1}) \end{aligned} \tag{1}$$

2.2 Mini-batch GNNs Training

Mini-batch training is a practical solution for scaling GNN training on very large graphs. Neighbor sampling is used to generate mini-batches, allowing sampling-based GNN models to handle unseen vertices. For example, GraphSAGE [16] samples multiple hops of neighbors for training as shown in Figure 1. The workflow of GraphSAGE training follows a vertex-centric computation paradigm including the following steps: 1, selecting a mini-batch of training vertices from the training set. 2, uniformly sampling the multiple hops of fixed-size neighbors for each training vertex. 3, extracting the

features of the sub-graph consisting of the training vertices and their neighbors to generate the mini-batch training data. Finally, performing *AGGREGATE* and *UPDATE* according to Equations 1, as well as executing the forward and backward propagation to update the model parameters.

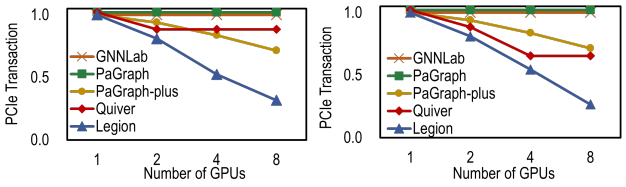
3 Observation and Motivation

When training large-scale graphs whose size exceeds the capacity of GPU memory on a multi-GPU server, the major performance bottleneck becomes the data movement from CPU to GPUs under the constraint of PCIe bandwidth. To this end, existing works [33, 42, 47] intend to relieve the PCIe bandwidth bottleneck by caching the hottest graph features on GPU memory. Though these cache-based approaches significantly reduce PCIe traffic, we still identify two issues of these existing cache-based GNN systems when training large-scale graphs: 1) poor multi-GPU cache scalability, and 2) coarse-grained GPU memory management for graph topology. In the following, we discuss each issue and the corresponding observation that motivates the design of Legion.

3.1 Multi-GPU Cache Scalability

As feature extraction occupies most of the data transferring from CPU to GPU, cache-based systems like GNNLab [47] maintain a global feature cache for vertices which are more frequently accessed via a pre-sampling phase. As training vertices are globally shuffled among all training GPUs, GNNLab replicates this cache across all GPUs involved in model training. Since a single GPU's memory space is quite limited, the fraction of cached features would inevitably become lower when the graph size grows, resulting in a lower cache hit ratio even on multi-GPU servers. To increase the cache capacity, the cache mechanism in Quiver [33] leverages high-speed NVLinks to support inter-GPU cache between NVLink-connected GPUs. Different from GNNLab, Quiver replicates feature cache between NVLink cliques and averagely hashes the features among GPUs in the same NVLink clique. However, this mechanism could still lead to poor cache scalability, especially when the NVLink clique is relatively small. E.g., the Siton server used in Table 1 has 4 NVLink cliques, each of which contains only 2 GPUs. Figure 2 illustrates that, in systems like Quiver, the PCIe transactions incurred by CPU-GPU data transferring stop decreasing when the number of GPUs is larger than the size of NVLink clique. This result shows that the cache performance in the above GNN systems cannot scale well with the increasing number of GPUs in modern servers.

To solve the scalability issue incurred by cache replication, PaGraph [23] partitions the graph in a self-reliance approach and maintains an independent cache for each partition using an in-degree-based metric on different GPUs. To train an L-layer GNN model, PaGraph extends every partition with redundant vertices and edges to include all the L-hop neighbor vertices for each train vertex in this partition. Each GPU only trains its own partition and synchronizes its local gradients periodically to update the model. However, the inclusion of the L-hop neighbor vertices leads to heavily duplicated cache contents on all GPUs. Figure 2 shows that the PaGraph exhibits a similar cache performance as GNNLab which adopts the cache replication mechanism. We further implement a PaGraph-plus design to alleviate the cache duplication issue in PaGraph. Specifically, we replace the graph partitioning algorithm in PaGraph with the XtraPulp [35] algorithm that minimizes edge-cuts between partitions and adopts a pre-sampling-based hotness metric to select the vertex features to be cached. Although PaGraph-plus achieves higher cache hit rates compared to PaGraph, the cache hit rates on different GPUs are very unbalanced as different partitions have various graph distributions. Figure 3 illustrates the load imbalance issue of PaGraph-plus by measuring the cache hit rates of eight GPUs. We observe that the hit rate varies by up to 17%.



(a) 2 GPUs per NVLink clique (b) 4 GPUs per NVLink clique

Figure 2: Comparing the cache scalability of cache-based GNN systems using the Products [17] dataset and 2-hop GraphSAGE [16] model in terms of normalized CPU-GPU PCIe transactions. The cache ratio is set to 5% $|V|$ on every GPU. The tested platforms are Siton (a) and DGX-V100 (b) servers, as shown in Table 1.

To sum up, for systems that globally shuffle the training vertices among GPUs in every iteration, such as GNNLab and Quiver, cache replication cannot be completely eliminated as each GPU may randomly access any vertex in the entire graph. Whereas the high-speed NVLinks between GPUs can be used to reduce the replication factor and expand the cache capacity. For systems that locally shuffle training vertices in each partition to produce mini-batches for different GPUs, such as PaGraph, the cache replication problem could be alleviated only when the model layer is small (e.g., less than 2). PaGraph-plus can further reduce cache duplication but faces another issue of unbalanced cache hit rates among GPUs.

Observation O1: Graph partitioning can be suitably guided by hardware structure. Different from Quiver, GNNLab, PaGraph, and PaGraph-plus do not take advantage of the NVLink between GPUs, which is a common capability in modern multi-GPU servers. As GPUs inside the same NVLink clique can access each other's memory via the low-latency high-throughput NVLink, an NVLink clique can hold the entire cache of a partition, which can be randomly sliced

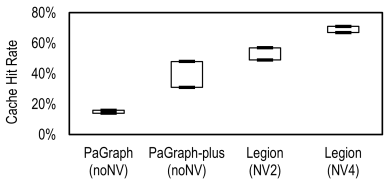
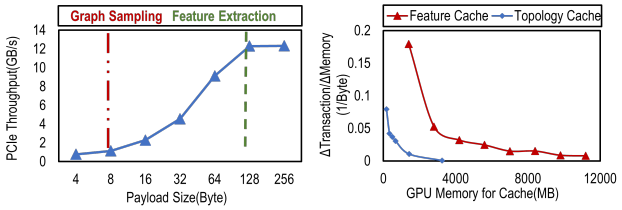


Figure 3: Cache hit rates of different systems in a server with 8 GPUs. The cache ratio is set to 5% $|V|$ on every GPU. The graph sampling follows the 2-hop GraphSAGE [16] model's setting using the Products [17] dataset. "NVx" means utilizing NVLink clique with x GPUs.



(a) PCIe throughput vs. payload size (b)

Figure 4: (a) The PCIe 3.0 throughput under different payload sizes of PCIe requests. (b) The PCIe traffic reduction rate for Paper100M with the growth of the cache capacity. The cache is on a single GPU and selected after pre-sampling.

and averagely allocated among GPUs inside a clique. This hardware-coherent design can balance the cache hit ratios between intra-clique GPUs. As the number of partitions is reduced to the number of NVLink cliques, it is more likely that the partitions follow a similar distribution (see the cache hit rate distribution of Legion in Figure 3). Inspired by **O1**, we propose an NVLink-aware hierarchical partitioning to preserve multi-GPU cache scalability in Legion (Section 4.1).

3.2 Coarse-grained GPU Memory Management for Graph Topology

In multi-GPU servers, the throughput of CPU-based graph sampling may not catch up with the throughput of GPU-based training. To improve the end-to-end training throughput, recent GNN systems [33,42,47] adopt GPUs to accelerate graph sampling. We observe that all these systems apply a very coarse-grained memory management mechanism for graph topology. In particular, they store the entire graph topology either in CPU memory or in a single GPU, depending on the size of graph topology: the graph topology is stored in CPU memory when it is too large or exceeds the capacity of a single GPU. The approach of storing the entire graph topology in a single GPU sets a hard limit on the scale of the graph. For example, a V100 GPU with 16GB memory can store at most 4 billion edges [16] without considering any other memory usage of feature cache and model training. When storing the graph topology in CPU memory, GPUs can directly access the graph topology via a unified virtual memory address (UVA [27]) technique. While the data access pattern of graph sampling is usually random and fine-grained. E.g., Figure 4a shows that the PCIe throughput of graph sampling is much lower than feature extraction. A large number of sampling PCIe transactions with small payload sizes will increase the CPU-GPU PCIe contention and lead to low bandwidth utilization.

Observation O2: The access of graph topology is skewed as graph features. Existing cache-based GNN systems [23, 33, 47] only maintain feature cache in GPU to reduce the CPU-GPU communication costs. However, we observe that the performance gain of the per-unit feature cache decreases once the cache capacity exceeds a threshold (see Figure 4b). We observe that the access of graph topology during graph sampling is also skewed as the access of features. Instead of allocating all the available GPU memory (except for the reservation for model training) for feature cache, it is reasonable to cache a subset of graph topology, i.e., edges of vertices that are frequently accessed during sampling, in the GPU memory to accelerate GPU sampling. Figure 4b shows that a relatively small topology cache can obviously reduce the number of PCIe transactions incurred by GPU sampling. Motivated by **O2**, we propose a hotness-aware unified cache in Legion. Specifically, Legion caches both graph topology and graph features with the goal of minimizing CPU-GPU communication overhead (see Section 4.2). Under the capacity limit of GPU memory, it is difficult to manually decide the optimal fractions of topology and feature cache. Legion solves this challenge with an automatic cache management mechanism, which can generate the optimal cache plan without requiring knowledge of hardware specifications from users.

4 Design of Legion

In order to address the aforementioned performance issues of existing cache-based GNN systems, we propose Legion, a cache-optimal GNN system that can push the envelope of the multi-GPU system automatically for billion-scale GNN training. The overall design of Legion is presented in Figure 5. We propose an NVLink-aware hierarchical partitioning technique (Section 4.1) in Legion that facilitates scaling up the cache capacity and reducing cache duplication in multi-GPU servers. To utilize GPU cache for both graph sampling and feature extraction, we present a hotness-aware unified cache (Section 4.2) that maintains both the topology and feature caches to optimize the overhead of PCIe traffic. We also develop an automatic cache management mechanism (Section 4.3) to automatically decide the memory allocations for both topology and feature caches.

4.1 NVLink-aware Hierarchical Partitioning

Motivated by **observation O1**, we propose a simple yet effective graph partitioning mechanism, referred to as **hierarchical**

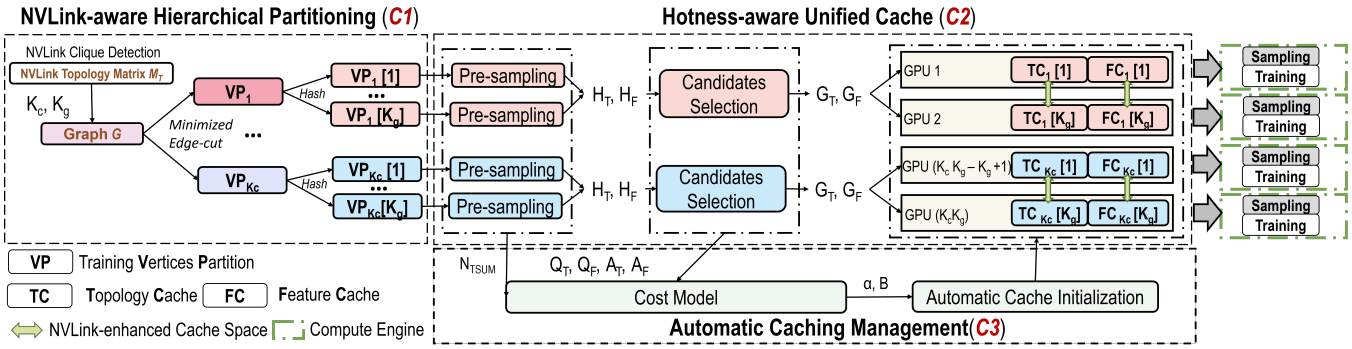


Figure 5: Design overview of Legion. Legion consists of three main contributions C1, C2, and C3.

partitioning, to facilitate cache scalability in Legion. Different from conventional graph partitioning algorithms which partition all edges/vertices of a graph into multiple tablets, hierarchical partitioning in Legion aims to divide the training vertices/edges into multiple disjoint tablets. The inputs of hierarchical partitioning are an NVLink topology matrix M_T of the underlying multi-GPU server and a graph G. The output is an assignment plan disseminating training vertices/edges among GPUs. Specifically, the process of hierarchical partitioning mainly consists of four steps:

S1: NVLink Clique Detection. With the topology matrix M_T of the server, Legion employs a MaxCliqueDyn algorithm [45] to identify the NVLink clique sets in M_T, and outputs the number of NVLink cliques K_c and the number of GPUs in each clique K_g.

S2: Inter-clique Graph Partitioning. To reduce the cache duplication between NVLink cliques, Legion uses an edge-cut minimizing partitioning algorithm, e.g., METIS [21] and XtraPulp [35], to split the input graph G into K_c partitions, i.e., P_1, P_2, ..., P_{Kc}, such that nodes are balanced among partitions and inter-partition edge-cuts are minimized. The training vertex set in P_i is denoted as VP_i. As the training vertices are randomly selected from G, the training vertex sets of different partitions are almost of the equal size. The number of partitions is equal to the number of detected NVLink cliques, and each NVLink clique hosts the cache for a dedicated partition. This way, Legion can reduce the cache duplication between NVLink cliques and take advantage of cache locality within an NVLink clique.

S3: Intra-clique Training Vertex Partitioning. As GPUs within an NVLink clique can access each other's memory via low-latency high-throughput NVLink interconnect, hierarchical partitioning further hashes the training vertex set of each partition into K_g tablets, where K_g is the GPU number in a clique. E.g., VP_i is split into $VP_i[1]$ and $VP_i[2]$ if K_g equals 2. Each tablet is exclusively mapped to a GPU in the corresponding NVLink clique. We explain how to generate the cache for each training vertex tablet in Section 4.2.

S4: Training Vertex Assignment. Finally, Legion assigns training vertices of each tablet to a corresponding GPU as the batch seeds, which will then be shuffled locally to generate mini-batches for graph sampling and training.

As such, Legion provides better cache scalability and load balancing compared to existing systems. Figure 2 shows the cache performance of Legion improves with the increase of GPUs almost linearly. Figure 3 illustrates that Legion has smaller fluctuations in the cache hit rates on multi-GPU servers with NVLink cliques of various sizes.

4.2 Hotness-aware Unified Cache

Motivated by the **observation O2**, we propose a hotness-aware unified cache to cache both graph topology and graph features. In this Section, we introduce the detailed mechanism of the unified cache.

4.2.1 Cache Structure

The unified cache consists of two parts: the topology cache and the feature cache. In particular, the topology cache maintains out-edge neighbor IDs for each selected hot vertex in the format of a compressed sparse row (CSR). As for the feature cache, Legion stores the feature vectors of selected hot vertices in the format of a 2D array, where each row is the feature vector of a selected hot vertex. Note that, the selected vertices in the topology and feature caches could be different.

4.2.2 Cache Construction

The construction of the unified cache is divided into three steps: (1) pre-sampling, (2) cache candidate selection, and (3) cache initialization. All the GPUs/NVLink cliques perform these steps concurrently to construct their own unified cache.

S1: Pre-sampling. Similar to GNNLab [47], Legion adopts a pre-sampling phase[2] to estimate the hotness metrics of graph topology and feature data during the training phase. Once the process of hierarchical partitioning is completed, the training vertex tablet assigned to each GPU is determined, which is used as the input for pre-sampling. The output of pre-sampling includes two hotness matrices: topology hotness matrix H_T and feature hotness matrix H_F. Each matrix's row represents the GPU IDs within an NVLink clique, the

[2]During pre-sampling, graph topology is stored in the CPU memory.

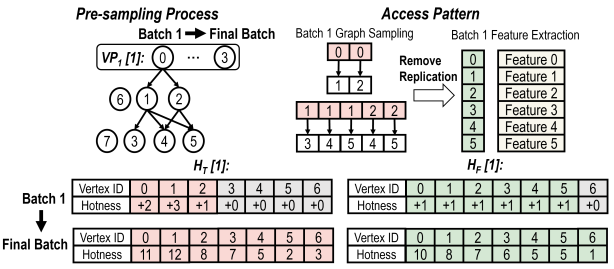


Figure 6: Update the hotness matrices of graph topology and features by pre-sampling. For simplicity, we only show the result for GPU 1.

column represents the vertex IDs, and the element H_{ij} of either matrix represents the hotness of the j-th vertex in the i-th GPU. During the pre-sampling, each GPU conducts a local shuffle on its own training vertex tablet to generate seeds for mini-batches, performs graph sampling for each mini-batch, and updates the corresponding row in H_T and H_F. Figure 6 shows a pre-sampling example. For H_T, whenever an edge is traversed during sampling, the hotness of its source vertex is incremented by 1. For H_F, the hotness for each vertex that appears in the sample results of the mini-batch is incremented by 1. Additionally, Legion uses Intel® Performance Counter Monitor (PCM) [18] to collect the summation of PCIe transactions number, N_{TSUM}, generated by all GPUs in an NVLink clique during pre-sampling.[3]

S2: Cache Candidate Selection. The objective of cache candidate selection is to select and disseminate the hot topology sub-structures and features among GPUs within the same NVLink clique based on pre-sampled hotness matrices. Thus this phase is conducted in the unit of NVLink clique, and each clique requires one GPU to perform the computation. The detailed process of cache candidate selection is presented in Algorithm 1. In brief, this algorithm computes the global topology/feature hotness of all vertices, i.e., A_T and A_F, in the NVLink clique by conducting a column-wise sum on H_T and H_F, respectively (Line 1). A_T and A_F are then sorted in descending order to generate Q_T and Q_F(Line 2). Next, We iterate Q_T and Q_F in order and assign every visited vertex to the GPU with the highest local hotness in H_T and H_F. For each GPU, we maintain two queues, i.e., G_T, G_F, whose order represents the priority of vertices to be included in this GPU cache. The outputs of Algorithm 1 are further used by the cost model (see Section 4.3) to generate the physical cache plan.

S3: Cache Initialization and Fill-up. Legion's cache management automatically decides the cache ratio for topology and feature so that the overall throughput is maximized (see Section 4.3). Guided by this mechanism, Legion allocates memory for both the topology and feature cache (TC and FC) of each GPU, and fetches the corresponding topology and feature data from CPU memory to fill up each GPU cache according to the corresponding cache orders in G_T and G_F.

[3] N_{TSUM} is further used by cost model's evaluation.

Algorithm 1 COMPLETE SHARING WITH LOCAL PREFERENCE (CSLP)

Input : K_g: number of GPUs per NVLink clique
H_F: feature hotness matrix
H_T: topology hotness matrix

Output : A_F: accumulated vertex-wise feature hotness vector
A_T: accumulated vertex-wise topology hotness vector
Q_T: vertex ID queue representing clique-level topology order,
Q_F: vertex ID queue representing clique-level feature order
G_T: vertex ID queue representing GPU-level topology order
G_F: vertex ID queue representing GPU-level feature order

/* Step 1: Accumulate each vertex's hotness from K_g GPUs. */
1 $A_F = H_F.columnWiseSum()$; $A_T = H_T.columnWiseSum()$;
/* Step 2: Sort vertices in A_F and A_T */
2 Q_F <- SortbyKeyDescend(A_F); Q_T <- SortbyKeyDescend(A_T);
/* Step 3: Assign each vertex to the GPU with the highest local hotness. */
3 **for** *v_id in* Q_T **do**
4 $gpu\_id = max(H_T[1:K_g][v\_id]).index$;
$G_T[gpu\_id].push(v\_id)$;
5 **end**
6 **for** *v_id in* Q_F **do**
7 $gpu\_id = max(H_F[1:K_g][v\_id]).index$;
$G_F[gpu\_id].push(v\_id)$;
8 **end**

4.3 Automatic Cache Management

The design of the unified cache poses a new challenge: how to properly specify the cache size for graph topology and features under the constraint of GPU memory such that the overall training throughput is maximized.

The general idea is to predict the overall throughput under different cache plans and search for the best cache plan that maximizes overall throughput. We define the cache plan as a cache memory management setting (B, α) at the NVLink clique granularity, where B is the multi-GPU cache memory size in an NVLink clique and α is the memory ratio for topology cache. B is identical among NVLink cliques and is by default set as the total multi-GPU memory minus the size of GPU memory reserved for GNN models and intermediate buffers in an NVLink clique. We need three steps to determine the optimal cache memory management setting (B, α), as discussed in Sections 4.3.1, 4.3.2, and 4.3.3.

4.3.1 Estimating Overall Throughput

The key goal of this Section is to build the relationship between the overall throughput and a cache plan. We build the relationship by estimating a key factor: the total PCIe traffic N_{total}, due to two reasons. First, the PCIe traffic is the major bottleneck of the overall system throughput, and lower PCIe traffic leads to higher overall system throughput. Second, varying cache plans major results in the variance of PCIe traffic.[4] Because each NVLink clique maintains caches for its own partition, we independently select the optimal cache plan for each NVLink clique so as to minimize the PCIe traffic of

[4] Though NVLink traffic is also influenced by the cache plan, we neglect it since NVLink has a much higher bandwidth than PCIe.

each NVLink clique. Thus, the overall system's PCIe traffic is minimized.

4.3.2 Cost Model to Estimate N_{total}

The key goal of this Section is to present a cost model to estimate N_{total} under a specific cache plan (B, α). First, given a specific cache plan (B, α), we can calculate the topology cache size m_T and the feature cache size m_F. Second, we find which vertices' topology/features should be stored in the topology/feature cache. Third, we estimate the PCIe traffic for graph sampling (N_T) and for feature extraction (N_F) with the current topology/feature cache utilization. At last, we estimate N_{total} by adding up N_T and N_F, as shown in Equation 2.

$$N_{total} = N_T + N_F \tag{2}$$

To estimate N_T and N_F, we need to collect other information apart from a given cache plan: the hotness vectors A_T and A_F, the summation of PCIe transaction number N_{TSUM} incurred by graph sampling, and the order queues of topology/feature cache candidates, Q_T and Q_F.

Estimating N_T. We estimate N_T when the memory size of a topology cache under one specific cache plan (B, α) is m_T, where $m_T = B \times \alpha$. The estimation consists of three steps.

First, with m_T, we decide which vertices' topology should be cached. We define V as the set of all vertices in the graph. And we define V_{Tcache} as the set of all vertices whose topology is cached under current topology cache size m_T. To get V_{Tcache}, we increase vertices and their topology into the cache with the growth of occupied topology cache memory by the order Q_T. Until the overall occupied topology cache memory size reaches m_T, we record V_{Tcache}. Equation 3 illustrates the relation between m_T and V_{Tcache}, where $nc(v)$ means the neighbor count of the vertex v. Here we assume the data types are Uint64 and Uint32 for the row and the column indices of the compressed sparse row format (CSR), respectively. We use s_{uint64} and s_{uint32} to denote the number of bytes to store a single Uint64 and Uint32 data accordingly.

$$\sum_{v \in V_{Tcache}} (nc(v) \times s_{uint32} + s_{uint64}) = m_T \tag{3}$$

Second, once we get V_{Tcache}, we can calculate the ratio of the PCIe transaction reduced by the topology cache by Equation 4. Let $a_T(v)$ mean the topology hotness of a specific vertex v ($a_T(v) \in A_T$).

$$R_T = \frac{\sum_{v \in V_{Tcache}} a_T(v)}{\sum_{v \in V} a_T(v)}) \tag{4}$$

Third, we get N_T by multiplying the entire PCIe transaction N_{TSUM} with the ratio of PCIe transactions that can **not** be reduced by the topology cache. We can get N_T by Equation 5.

$$N_T = N_{TSUM} \times (1 - R_T) \tag{5}$$

Estimating N_F. We explain how to calculate N_F when the feature cache memory size is m_F, where $m_F = B \times (1 - \alpha)$. There are also three steps in estimation.

First, given m_F, we decide which vertices' features should be cached. We define V_{Fcache} as the set of vertices whose feature data is cached. Then we increase the vertices with their feature into cache by the order Q_F until the occupied feature cache memory size reaches m_F, as defined in Equation 6. D represents the dimension of a feature vector and the feature data is the Float32 type each of which needs $s_{float32}$ bytes to store.

$$\sum_{v \in V_{Fcache}} D \times s_{float32} = m_F \tag{6}$$

Second, as shown in Equation 7, we calculate the total number of features U_F that still needs transferring through PCIe with a feature cache.

$$U_F = \sum_{v \in V} (a_F(v)) - \sum_{v \in V_{Fcache}} (a_F(v)) \tag{7}$$

Third, we get N_F by multiplying the transaction number needed by transferring one vertex's feature with the total number of features to be transferred, U_F, as shown in Equation 8. Here CLS means the transferred cache line size. CLS might be different for various CPUs and GPUs. We can get the CLS from PCM. E.g., CLS equals 64 in our machine settings. And $a_F(v)$ means the hotness of a specific vertex v ($a_F(v) \in A_F$).

$$N_F = (\lceil \frac{D \times s_{float32}}{CLS} \rceil) \times U_F \tag{8}$$

4.3.3 Searching for Optimal Cache Plan in Parallel

The key goal of this Section is to efficiently determine the optimal cache plan for each clique. As discussed in Section 4.3.1, we search for the optimal cache plan independently with one GPU for each NVLink clique. In each NVLink clique, we first need to traverse α from 0 to 1 by an interval $\Delta\alpha$ [5] to generate the candidate cache plans, and the calculate N_{total} accordingly. Then we need to search N_{total} sequences and find the smallest one with the dedicated α. To minimize overhead, the process is well parallelized, including four steps:

First, we generate all the candidate cache plans in parallel and get sequences of m_T and m_F in each setting.

Second, we get the boundaries of cached vertices set V_{Tcache} and V_{Fcache} using Equations 3 and 6, where the boundaries are the largest cached vertices' indexes in Q_T and Q_F. To do so, we get the topology and feature memory size of every single vertex in parallel and store them in two arrays, $S_{Tsingle}$ and $S_{Fsingle}$, following the vertices order, Q_T and Q_F. Next, we calculate the cumulative sum of $S_{Tsingle}$ and $S_{Fsingle}$ by a parallel inclusive scan and get S_{Tsum} and S_{Fsum}. Then for each cache plan with m_T and m_F, we use a parallel binary

[5] $\Delta\alpha$ is set to be 0.01 by default.

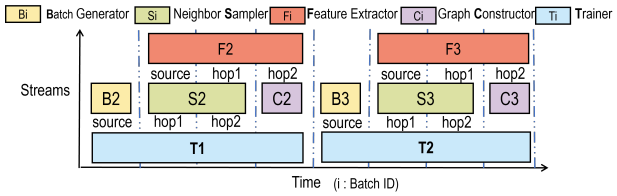


Figure 7: An example of fine-grained GNN training pipeline for 2-hop GraphSAGE model.

search towards S_{Tsum} and S_{Fsum} to get the boundary indexes of vertices, respectively.

Third, we get the R_T and U_F according to Equations 4 and 7. To do so, we calculate the cumulative sum of A_T and A_F by a parallel inclusive scan and get A_{Tsum} and A_{Fsum}. Then for each cache plan, we lookup A_{Tsum} and A_{Fsum} with the boundary indexes of vertices set V_{Tcache}, V_{Fcache}, and get $\sum_{v \in V_{Tcache}} a_T(v)$ and $\sum_{v \in V_{Fcache}} a_F(v)$, respectively. Similarly, after lookup the largest indexes in A_{Tsum} and A_{Fsum}, we get $\sum_{v \in V} a_F(v)$ and $\sum_{v \in V} a_F(v)$. As such, we can get the corresponding R_T and U_F.

At last, we calculate N_T and N_F for each cache plan according to Equation 5 and 8. Then we get N_{total} by Equation 2 and search in parallel for the smallest N_{total} with the corresponding α.

After getting the optimal cache plans (B, α), Legion can automatically allocate the cache space and fill up the cache.

5 Implementation of Legion

Legion mainly consists of two components, which are the sampling server and the training backend. The sampling server is implemented from scratch and the training backend is built on top of Pytorch [31]. The sampling server is responsible for generating sampled results, and the training backend takes the sampled results as input to train the GNN models.

In Legion, every GPU executes the graph sampling, feature extraction, and model training stages, and all these stages are scheduled in a fine-grained pipeline to fully utilize the GPU computation cycles. Figure 7 illustrates how the training process is pipelined for a 2-hop GraphSAGE [16] model. In order to improve the overall throughput, we design an inter-batch pipeline overlapping the tasks of the sampling server and the training backend for different batches. E.g., the training of batch B_i can be overlapped with the sampling and feature extraction of batch B_{i+1}. To further improve the throughput of sampling and feature extraction, we design an intra-batch pipeline inside the sampling server. Specifically, we break down the workloads of the sampling server into four types, each of which corresponds to a type of operator: (1) Batch generator shuffles the local training vertices to generate seeds for mini batches; (2) Neighbor sampler executes the L-hop neighbor sampling; (4) Feature extractor extracts the feature of the batch seeds and vertices in the sampled results; (4) Graph constructor is used to generating the subgraph based on the sampled results. For the same batch, graph sampling and graph construction can be overlapped with feature extraction.

Table 1: GPU Server Statistics.

Server	DGX-V100	Siton	DGX-A100
GPU Type	16GB-V100x8	40GB-A100x8	80GB-A100x8
NVLink Topo.	$K_c = 2, K_g = 4$	$K_c = 4, K_g = 2$	$K_c = 1, K_g = 8$
PCIe Gen.	3.0x16	4.0x16	4.0x16
PCIe Topo.	4 switches, 2 GPUs/switch	2 switches, 4 GPUs/switch	4 switches, 2 GPUs/switch
CPU Mem.	384GB	1TB	1TB
CPU Core Num.	96	104	128
Sockets, NUMA Num.	2, 1	2, 2	2, 1

Table 2: Dataset Statistics.

Dataset	PR	PA	CO	UKS	UKL	CL
Vertices	2.4M	111M	65M	133M	0.79B	1B
Edges	120M	1.6B	1.8B	5.5B	47.2B	42.5B
Topology Storage	640M	6.4GB	7.2GB	22GB	189GB	170GB
Feature Size	100	128	256	256	128	128
Feature Storage	960M	56GB	65GB	136GB	400GB	512GB

6 Evaluation

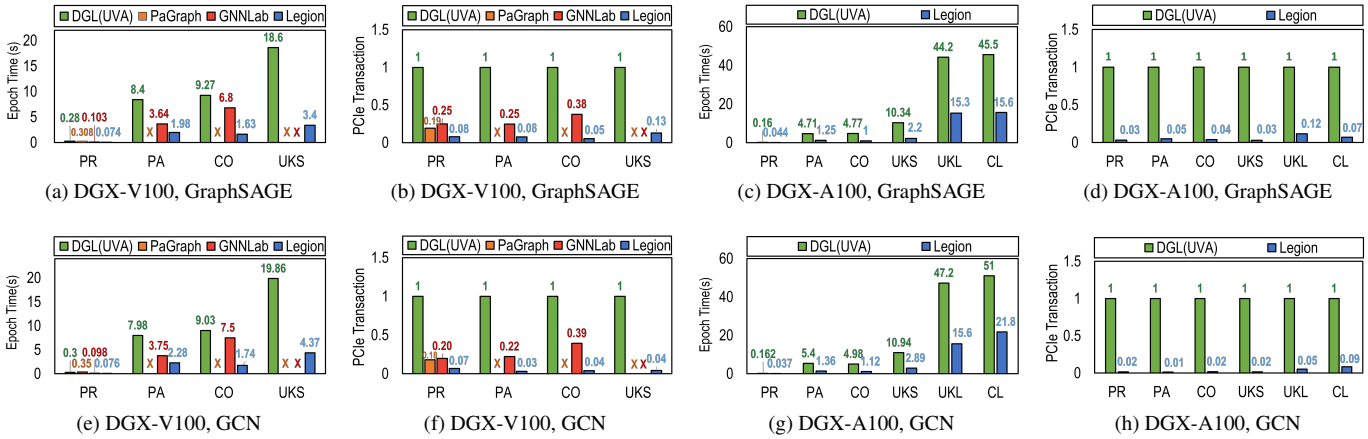
6.1 Experimental Setting

Experimental Platform. The experiments are conducted using three different GPU servers: DGX-V100, Siton, and DGX-A100, as shown in Table 1. For DGX-A100, we set the upper limit of GPU memory to 40 GB.

GNN Models. We use two sampling-based GNN models: GraphSAGE [16] and GCN [22], which both adopt a 2-hop random neighbor sampling. The sampling fan-outs are 25 and 10. The dimension of the hidden layers in both models is set to 256. Similar to existing work [47], the batch size is set to 8000. Unless explicitly explained, node classification is used as the GNN task.

Datasets. We conduct our experiments on multiple real-world graph datasets with various scales. Table 2 shows the dataset characteristics. The Products (PR) and Paper100M (PA) are available in Open Graph Benchmark [17]. The Com-Friendster (CO) graph is an online gaming network [46]. And the Uk-Union (UKS), UK-2014 (UKL), and Clue-web (CL) are from WebGraph [2–5]. As CO, UKS, UKL, and CL have no feature, we manually generate the features with the dimension specified as 128 or 256. Following PR's setting, we choose 10% of vertices from each graph as training vertices.

Baselines. We use DGL [42], PaGraph [23] and GNNLab [47] as the baseline systems. The DGL version is v0.9.1, which supports accessing graph topology and features via the UVA technique. We don't compare with Quiver [33] in the overall performance experiment as its open-sourced version cannot support training on servers with 8 GPUs. Instead,



(a) DGX-V100, GraphSAGE (b) DGX-V100, GraphSAGE (c) DGX-A100, GraphSAGE (d) DGX-A100, GraphSAGE

(e) DGX-V100, GCN (f) DGX-V100, GCN (g) DGX-A100, GCN (h) DGX-A100, GCN

Figure 8: Overall performance of Legion comparing with state-of-the-art systems. "×" denotes OOM (out of memory).

we implement a Quiver-like multi-GPU cache mechanism in Legion for comparison in Section 6.3.

6.2 End-to-end Performance

We compare the end-to-end performance of Legion with baseline systems on the DGX-V100 and DGX-A100 servers. On the DGX-V100 server, we evaluate PR, PA, CO, and UKS graphs whose graph topology and features can fit into 384 GB CPU memory. On the DGX-A100 server, we evaluate all six graphs. As PaGraph and GNNLab are implemented using CUDA 10 which cannot support A100 GPU, we exclude them from the experiments using DGX-A100.

Baseline Configuration. For all the baselines, we manually adjust their configurations to achieve optimal performance. DGL uses the UVA mode, where sampling is performed in GPU, and the topology and features are all stored in CPU memory. The number of worker threads in PaGraph is set to be 64 to maximize the CPU sampling throughput. For GNNLab, we adjust the numbers of sampling and training GPUs such that the overall throughput is maximized. In contrast, Legion relies on its automatic cache management mechanism to generate the unified cache plan.

Evaluation Metrics. We record the average epoch time for all systems. We also use PCM [18] to measure the maximum PCIe counter value across different sockets and report the normalized values based on the result of DGL for all systems.

Support training on large graphs. As shown in Figures 8a, 8e, 8c and 8g, Legion outperform all the baseline systems in every setting. Specifically, Legion achieves 3.78-5.69× speedup for GraphSAGE (3.5-5.19× for GCN) on DGX-V100 and 2.89-4.77× speedup for GraphSAGE (2.34-4.45× for GCN) on DGX-A100 over DGL(UVA). Figures 8b, 8f, 8d and 8h show that, compared with the baselines, Legion can sufficiently utilize the multi-GPU cache to minimize PCIe traffic incurred by CPU-GPU data transferring. GNNLab runs out of GPU memory for UKS on DGX-V100 as the size of graph topology exceeds the capacity of single GPU memory. PaGraph runs out of the CPU memory for most graphs except for PR on DGX-V100, as the memory management in PaGraph incurs extra memory overheads, including duplicated multi-hop neighbors in CPU memory and redundant intermediate buffers generated during computation.

Speedup over SOTA system on small graphs. Legion achieves 1.39-4.18× speedup for GraphSAGE (1.29-4.32× speedup for GCN) over GNNLab on the small graphs (PR, PA, CO). The performance gain mainly comes from two aspects. First, Figure 8b and 8f show that Legion significantly reduces the PCIe traffic for PA and CO, as it has a scalable multi-GPU cache design compared with GNNLab. The reduction of PCIe traffic relieves the CPU-GPU communication bottleneck such that the overall performance is improved. Second, Legion can use all GPUs for model training, while GNNLab needs to allocate several GPUs for sampling exclusively due to its factored design. In Legion, the graph sampling is overlapped by model training due to the fine-grained pipeline (see Section 5). E.g., when training GraphSAGE using the PR dataset, all the topology and feature data can be stored in GPU memory in both Legion and GNNLab. However, Legion can use 8 GPUs for training while GNNLab only uses 4 GPUs for training (see Figures 8a).

6.3 Effect of Hierarchical Partitioning

In this experiment, we examine the effect of hierarchical partitioning in Legion. We report the cache hit rates under different partition strategies in all three GPU servers: DGX-V100 (NV4: $K_c = 2$ and $K_g = 4$), Siton (NV2: $K_c = 4$ and $K_g = 2$) and DGX-A100 (NV8: $K_c = 1$ and $K_g = 8$).

6.3.1 Cache Performance

Baselines. For a fair comparison, we implement the cache designs of GNNLab, PaGraph-plus (described in Section 3.1), and Quiver-plus in Legion and compare their cache hit rates. Specifically, GNNLab maintains a globally replicated cache among all GPUs without using NVLinks (noPart+noNV).

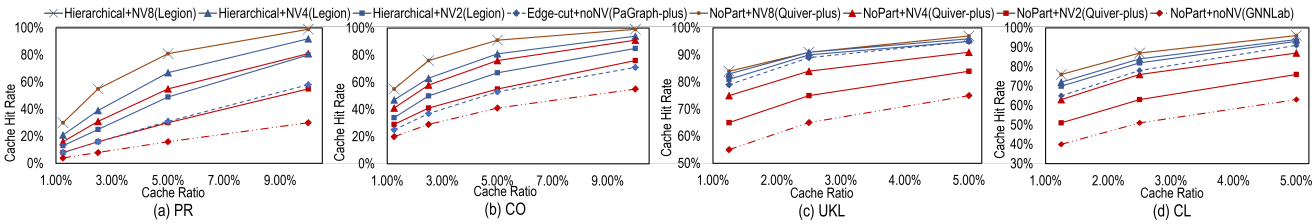


Figure 9: Effect of graph partition strategies (NoPart: no partitioning; Edge-cut: partitioning minimizing edge-cut; Hierarchical: hierarchical partitioning) to multi-GPU cache in terms of cache hit rate, with different NVLink infrastructures. (noNV: disable NVLinks; NV2: $K_c = 4$ and $K_g = 2$; NV4: $K_c = 2$ and $K_g = 4$; NV8: $K_c = 1$ and $K_g = 8$;).

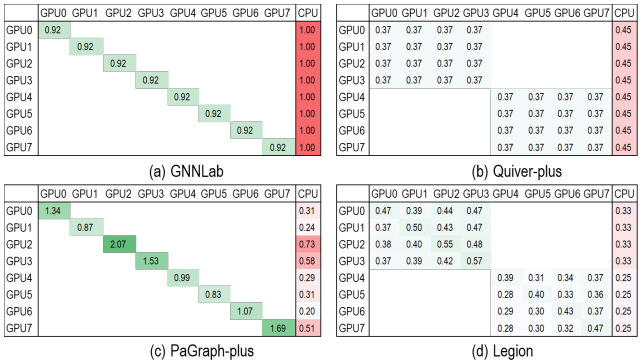


Figure 10: Data transferring in feature extraction of PA dataset on DGX-V100 (NV4). The rows and columns of each matrix denote the destination and source of data transferring. The right-most (red) column records the data transferring volume from CPU to GPU via PCIe. The middle (green) columns represent the GPU-GPU data transferring volume. We normalize the recorded values based on the CPU-GPU data transferring volumes in GNNLab.

Quiver-plus enables NVLink and maintains replicated cache among NVLink cliques (noPart+NV2 / noPart+NV4 / noPart+NV8). PaGraph-plus takes the XtraPulp [35] partitioning which minimizes across-partition edge-cuts and disables NVLinks (Edge-cut+noNV). Legion uses hierarchical partitioning (inter-NVLink-clique partitioning: XtraPulp) and enables NVLink (Hierarchical+NV2 / Hierarchical+NV4 / Hierarchical+NV8). We use the pre-sampling hotness metric for all these cache designs. The in-degree-based hotness metric in the original PaGraph and Quiver design are replaced with the pre-sampling hotness metric in Pagraph-plus and Quiver-plus, which has a better performance on cache hit rates [47].

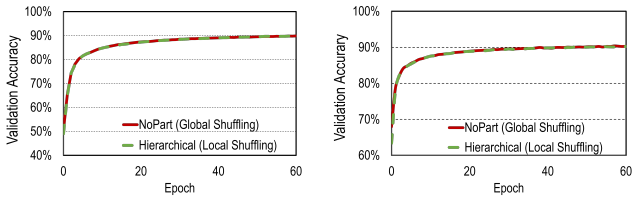
The datasets used in this experiment are PR, CO, UKL, and CL. We vary the cache ratio from 1.25% $|V|$ to 10% $|V|$ for PR and CO. For UKL and CL whose sizes are relatively large, the cache ratio varies from 1.25% $|V|$ to 5% $|V|$. Figure 9 shows that, for almost all the experiment settings, Legion has the highest cache hit rate. Specifically, Legion obviously outperforms Quiver-plus in the cases of NV2 and NV4, since Legion can reduce the inter-NVLink-clique cache duplication and achieves higher multi-GPU memory utilization compared with Quiver-plus. For the case of NV8, as all GPUs are in the same NVLink clique, the inter-clique graph partitioning in Legion can be skipped, and hierarchical partitioning turns into hash partitioning among all the GPUs, which is identical to Quiver-plus in the case of NV8. Legion outperforms PaGraph-plus because it has much less cache duplication. Specifically, PaGraph-plus's cache mechanism may replicate vertices with high global hotness on multiple GPUs. Compared with GNNLab, Legion has higher cache hit rates as it can scale up the cache capacity with the increase of GPUs, while GNNLab replicates the same feature cache across all GPUs. These results demonstrate that Legion can effectively adapt the cache plan to optimize the cache performance for multi-GPU servers with various NVLink topologies.

6.3.2 Data Transferring in Feature Extraction

In this experiment, we demonstrate the GPU-GPU and CPU-GPU data transferring volume during feature extraction using the PA dataset. Specifically, we perform the graph sampling and feature extraction stages using the PA graph on DGX-V100 (NV4) and record the data transferring volumes of feature extraction on each GPU in the format of a traffic matrix. We use GNNLab, PaGraph-plus, and Quiver-plus as the baselines, and set the feature cache ratio on each GPU to 2.5% $|V|$. The results are presented in Figure 10. We can see that Legion's data transferring volume from CPU to GPU is the smallest, indicating the best cache performance among the compared systems. As it is the GPU with the largest CPU-GPU data transferring volume that dominates the overall performance, although Legion's CPU-GPU volumes on some GPUs are higher than PaGraph-plus, Legion can still outperform PaGraph-plus because its largest CPU-GPU volume is lower than that of PaGraph-plus.

6.3.3 Model Convergence

Compared with global shuffling (randomly generating batch seeds from the vertex set of the entire graph), recent studies [23, 28] show that local shuffling (generating batch seeds within partitions) brings negligible impact on the rate of model convergence. Legion adopts local shuffling, and we conduct an experiment on the Siton server (NV2) to compare



(a) GraphSAGE (b) GCN

Figure 11: Comparing local shuffling and global shuffling on model convergence (NoPart: no partitioning; Hierarchical: hierarchical partitioning).

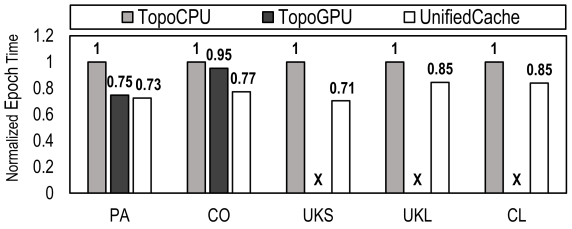


Figure 12: The impact of topology cache. "×" means OOM (out of memory).

its convergence speed with global shuffling on both GraphSAGE and GCN using the PR dataset. The results in Figure 11 show that the local shuffling of Legion could catch up with the convergence speed of global shuffling.

6.4 Effect of Unified Cache

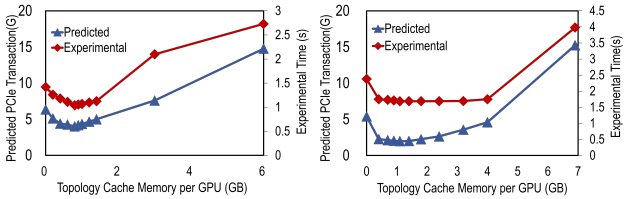
Different from existing cache-based systems, Legion's unified cache also takes graph topology into account. In this experiment, we demonstrate the benefits of topology cache.

We compare the training epoch time of unified cache in Legion with two baselines: (1) storing all topology in the CPU (denoted as TopoCPU) and (2) replicating the entire topology in every single GPU (denoted as TopoGPU). For a fair comparison, we implement both TopoCPU and TopoGPU in Legion and use the same GPU memory volume for the three settings. Among the three settings, TopoCPU has the most GPU memory available for the feature cache, and the TopoGPU has the least GPU memory for the feature cache or even runs out of GPU memory. We evaluate PA, CO, and UKS on DGX-V100 and evaluate UKL and CL on DGX-A100.

As shown in Figure 12, the unified cache outperforms the other two baselines for all graphs. This result demonstrates that, when the size of the feature cache exceeds a threshold, the increase of cache hit rate slows down. In this case, caching some hot topology data in GPU memory will save the system from severe PCIe contention incurred by graph sampling and benefit the overall GNN training throughput.

6.5 Evaluation of Cost Model

Legion proposes the cost model to guide allocating GPU memory for both graph topology and feature cache. In this experiment, we evaluate the effectiveness of this mechanism.



(a) PA, Single GPU (b) UKS, DGX-V100 (NV4)

Figure 13: Evaluation of cost model. The left y-axis means the PCIe transaction number predicted by the cost model. The right y-axis represents the experimental per-epoch graph sampling and feature extraction time.

Table 3: Evaluation of Partitioning Cost.

Dataset	PA (DGX-V100)	UKL (Siton)
Graph Partition(min)	7.2	75
Data Loading From Disk To Memory(min)	0.32	3.5
Node Classification Epoch(s)	1.98	15.6
Link Prediction Epoch(min)	49.8	402

Specifically, we compare the predicted PCIe traffic with the experimental per-epoch execution time of graph sampling and feature extraction. In the experiment using the PA dataset, the GPU memory allocated for the cache is 10 GB. And in the experiment using the UKS dataset, the GPU memory allocated for the cache is 8 GB. When varying the size of the topology cache, the size of the feature cache is adjusted accordingly. Figure 13 shows that our cost model can precisely predict the trend of per-epoch execution time without manual interference.

6.6 Partitioning Cost

In this experiment, we study the partitioning cost in Legion. We run our experiment on the UKL dataset that has the largest number of edges among all the datasets, resulting in the highest cost of edge-cut partitioning. We also present the results of the PA data (medium size) to show the partitioning costs of different graph scales. We partition PA on DGX-V100 and UKL on Siton using the XtraPulp algorithm. For node classification, we set the training set to be 10% of the total edges for both graphs. For link prediction, we set the training set to be 80% of total edges. When the graph is too large to be partitioned in memory, like UKL, we randomly sample a fraction of edges (25% for UKL) and keep all vertices in the graph such that the subgraph can be partitioned in memory. This technique can obviously speedup graph partitioning and preserves a low edge-cut ratio.

Table 3 shows the preprocessing cost of Legion's hierarchical partitioning. We observe that the partitioning cost is tolerable, because 1) we only partition the graph once but can use the partitioning results for multiple GNN training jobs, and 2) the GNN task like link prediction needs multiple epochs to converge while a single epoch often costs a long time to finish.

7 Related Work

To our knowledge, Legion is the first work that automatically pushes the envelope of multi-GPU systems for billion-scale GNN training. In the following, we contrast Legion and existing works in the following aspects.

GNN Frameworks. Several GNN systems [11, 12, 20, 23, 26, 33, 38, 42, 43, 47, 51, 53, 55] have emerged in recent years. Most of these GNN systems are built on top of deep learning frameworks like Pytorch [31], TensorFlow [1] and MXNet [9].

GPU Sampling. NextDoor [19] and C-SAW [30] focus on accelerating GPU sampling kernel. DGL [42] also supports GPU sampling in its recent release. Quiver [33] can support GPU sampling with the entire topology either stored in the single GPU or in the CPU memory. GNNLab [47] adopts a factored design where each GPU is dedicated to graph sampling or model training exclusively. In contrast, Legion uses all GPUs for end-to-end GNN acceleration.

Graph Partitioning. Graph partitioning such as [6, 14, 15, 21, 32, 35, 36, 39], has been widely adopted in GNN systems. DGL [42] adopts METIS [21] to partition the graph. PaGraph [23] adopts a self-reliant partitioning strategy with the goal of achieving balanced training vertex allocation across GPUs and improving data locality on every GPU. DGCL [7] adopts a partitioning algorithm to partition the graph's physical edges and features and store them among distributed machines. In contrast, Legion adopts hierarchical partitioning to automatically partition graphs to each GPU in a single multi-GPU server accordingly to GPU interconnections.

GPU Feature Cache. PaGraph [23], BGL [24], GNNLab [47], Quiver [33] and [29] explore feature caching on GPU to accelerate GNN training. PaGraph [23] and Quiver [33] use the in-degree of vertexes as the hotness metric. BGL [24] applies a FIFO dynamic cache policy and selects training vertices in a BFS order for a higher cache hit rate, but hinders model convergence and incurs cache replacement overheads. [29] uses a weighted reverse PageRank algorithm as a hotness metric. GNNLab [47] uses vertices' access frequencies in the pre-sampling epoch as a hotness metric. In contrast, Legion automatically caches both features and topology with the highest hotness. And Legion statically partitions the graph with minimal edge-cut to preserve intra-partition data locality. Figures 9 and 11 show that Legion can achieve a high cache hit rate even with small cache ratios without compromising the model convergence rate.

Large Graph Systems. SSD-based GNN systems [41] and distributed GNN systems [12, 24, 52, 54] also aim at large-graph training and propose distinct approaches to solve I/O problems at various levels. MariusGNN [41] minimizes I/O between SSD and CPU by including valid graph data in a single swap as much as possible. Systems like BGL [24], DistDGLv2 [54], and P3 [12] optimize network I/O between distributed machines, whose network performance can be improved when introducing GPU-centric SmartNIC [44]. In contrast, Legion focuses on utilizing GPU caches to minimize PCIe traffic from CPU memory to multiple GPUs, which is orthogonal to the above systems.

8 Conclusion

We present Legion, a system that automatically pushes the envelope of multi-GPU systems for billion-scale GNN training. Legion has three key innovations. First, we propose an NVLink-aware hierarchical partitioning technique that helps minimize cache replication and extends the threshold of cache capacity beyond the limit of a single GPU or NVLink clique. Second, we propose a novel hotness-aware unified cache mechanism that helps accelerate both graph sampling and feature extraction. Third, we present an automatic cache management mechanism enabling optimal cache planning without requiring extra knowledge of hardware specifications and GNN performance details from users. Experiments show Legion outperforms SOTA cache-based GNN systems up to 4.32× and supports training on billion-scale graphs. And Legion is open-sourced at https://github.com/RC4ML/Legion.

Acknowledgements. We thank our shepherd Anand Iyer and anonymous reviewers for their detailed feedback. The work is supported by the following grants: the Program of Zhejiang Province Science and Technology (2022C01044), a research grant from Alibaba Group through the Alibaba Innovative Research (AIR) Program, the Fundamental Research Funds for the Central Universities 226-2022-00151, Key Laboratory for Corneal Diseases Research of Zhejiang Province, Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (SN-ZJU-SIAS-0010). Zeke Wang and Fei Wu are the corresponding authors.

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 Levenberga, Sherry Moore Rajat Monga, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Martin Wicke Pete Warden, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *OSDI*, 2016.

[2] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 2004.

[3] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. Bubing: Massive crawling for the masses. In *WWW*, 2014.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, 2011.

[5] Paolo Boldi and Sebastiano Vigna. The web graph framework: Compression techniques. In *WWW*, 2004.

[6] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *SC*, 2013.

[7] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. DGCL: An efficient communication library for distributed GNN training. In *Eurosys*, 2021.

[8] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[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. *arXiv preprint arXiv:1512.01274*, 2015.

[10] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *SIGKDD*, 2019.

[11] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[12] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *OSDI*, 2021.

[13] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *ICML*, 2017.

[14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[15] 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 *OSDI*, 2014.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *NeurIPS*, 2017.

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *NIPS*, 2020.

[18] Intel. PCM. https://github.com/intel/pcm, 2022.

[19] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In *Eurosys*, 2021.

[20] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *MLSys*, 2020.

[21] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[22] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[23] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *SoCC*, 2020.

[24] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021.

[25] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. Pick and choose: a gnn-based imbalanced learning approach for fraud detection. In *WWW*, 2021.

[26] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *USENIX ATC*, 2019.

[27] Mark Harris. Unified Memory for CUDA Beginners. ://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2017.

[28] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 2019.

[29] Seung Won Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-mei Hwu. Graph neural network training and data tiering. In *SIGKDD*, 2022.

[30] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. C-saw: A framework for graph sampling and random walk on gpus. In *SC*, 2020.

[31] 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. *NeurIPS*, 2019.

[32] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *CIKM*, 2015.

[33] QuiverTeam. Quiver. https://github.com/quiver-team/torch-quiver, 2021.

[34] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *ASPLOS*, 2023.

[35] George M Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.

[36] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, 2012.

[37] Chang Su, Yu Hou, and Fei Wang. Gnn-based biomedical knowledge graph mining in drug development. In *Graph Neural Networks: Foundations, Frontiers, and Applications*. 2022.

[38] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Dorylus: Affordable, scalable, and accurate nn training with distributed cpu servers and serverless threads. In *OSDI*, 2021.

[39] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*, 2014.

[40] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[41] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *Eurosys*, 2023.

[42] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR*, 2019.

[43] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for gnnacceleration on gpus. In *OSDI*, 2021.

[44] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. FpgaNIC: An FPGA-based versatile 100gb SmartNIC for GPUs. In *ATC*, 2022.

[45] Wikipedia. MaxCliqueDyn. https://en.wikipedia.org/wiki/MaxCliqueDyn_maximum_clique_algorithm, 2022.

[46] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *SIGKDD*, 2012.

[47] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: A factored system for sample-based gnn training over gpus. In *Eurosys*, 2022.

[48] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *SIGKDD*, 2018.

[49] Zhongbao Yu, Jiaqi Zhang, Xin Qi, and Chao Chen. Application research of graph neural networks in the financial risk control.

[50] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[51] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, and Yuan Qi. Agl: a scalable system for industrial-purpose graph machine learning. *arXiv preprint arXiv:2003.02454*, 2020.

[52] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2020.

[53] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *SIGKDD*, 2022.

[54] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, Qidong Su, Minjie Wang, Chao Ma, and George Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale graphs. *arXiv preprint arXiv:2112.15345*, 2021.

[55] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *VLDB*, 2019.

A Appendices

A.1 Generalization of Legion

Generalizing Legion to SSDs. Legion is primarily designed for in-memory graph training, but it can also be extended to SSD-based systems. First, Legion can still use GPU to execute end-to-end GNN training while storing graph topology and features in SSDs. The data path between GPU and SSDs could be enabled by BaM [34], which is a GPU-initiated on-demand high-throughput storage access technique. Second, the throughput of reading data from SSDs could be much lower than that in memory, leading to more severe I/O problems. Legion's hierarchical partitioning and unified cache design could still help reduce I/O and benefit overall throughput in this situation. Finally, due to the limited GPU memory, there still exists a trade-off between topology cache and feature cache in SSD-based systems. Thus automatic cache management could still be important and should be extended with more considerations of the specific hardware characteristics. We leave Legion's generalization to SSDs as our future work.
Generalizing Legion to none-NVLink systems. Legion can still bring performance benefits in multi-GPU systems without NVLink. To achieve this, Legion splits the graph into N partitions and each GPU maintains a cache for a partition. This approach can have a higher cache hit rate compared to replicating a global cache, as shown in Figure 9. We can also apply Legion on other GPU platforms, e.g., on AMD GPUs by leveraging the AMD Infinity inter-GPU bus.

Bridging the Gap between Relational OLTP and Graph-based OLAP

Sijie Shen[1,2], Zihang Yao[1], Lin Shi[1], Lei Wang[2], Longbin Lai[2], Qian Tao[2], Li Su[2], Rong Chen[1,3], Wenyuan Yu[2], Haibo Chen[1], Binyu Zang[1], and Jingren Zhou[2]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[2]Alibaba Group
[3]Shanghai AI Laboratory

Abstract

Recently, many applications have required the ability to perform dynamic graph analytical processing (GAP) tasks on the datasets generated by relational OLTP in real time. To meet the two key requirements of performance and freshness, this paper presents GART, an in-memory system that extends hybrid transactional/analytical processing (HTAP) systems to support GAP, resulting in hybrid transactional and graph analytical processing (HTGAP). GART fulfills two unique goals that are not encountered by HTAP systems. First, to adapt to rich workloads flexibility, GART proposes transparent data model conversion by graph extraction interfaces, which define rules for relational-graph mapping. Second, to ensure GAP performance, GART proposes an efficient dynamic graph storage with good locality that stems from key insights into HTGAP workloads, including (1) an efficient and mutable compressed sparse row (CSR) representation to guarantee the locality of edge scan, (2) a coarse-grained multi-version concurrency control (MVCC) scheme to reduce the temporal and spatial overhead of versioning, and (3) a flexible property storage to efficiently run different GAP workloads. Evaluations show that GART performs several orders of magnitude better than existing solutions in terms of freshness or performance. Meanwhile, for GAP workloads on the LDBC SNB dataset, GART outperforms the state-of-the-art general-purpose dynamic graph storage (i.e., LiveGraph) by up to 4.4×.

1 Introduction

Graphs, due to their natural ability to model intricate relations among entities [12, 61], have been intensively adopted to model business data. Correspondingly, *graph analytical processing (GAP)* techniques are being developed to better understand graph data and are widely applied in many fields, such as recommendation systems [70, 74], supply-chain analysis [46], and fraud detection [29, 56]. As business data is constantly generated and updated, there calls for an urgent need for *dynamic* GAP workloads on *real-time* datasets. In many traditional business scenarios, data is usually updated by online transaction processing (OLTP) in relational databases [26, 47, 82].

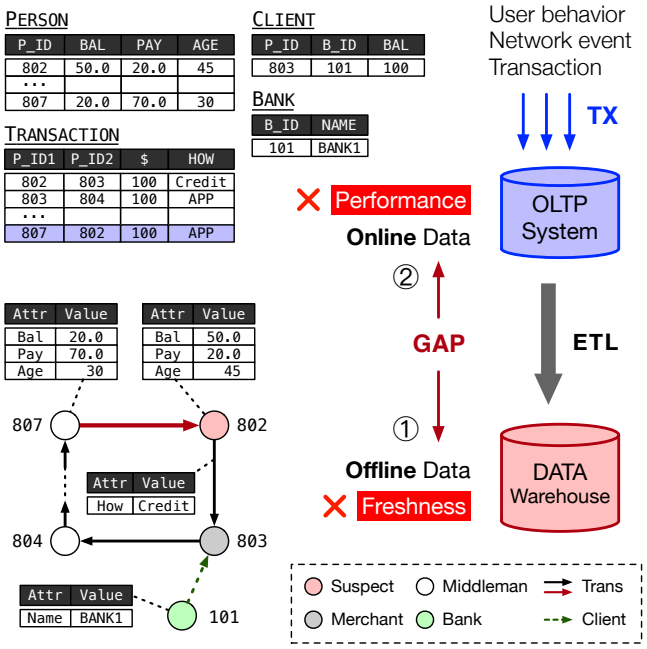


Fig. 1. *A comparison of solutions for dynamic graph analytical processing* (GAP) *on transactional datasets and their limitations.* **Solutions***: graph processing on offline data* (①) *or online data* (②).

Real-world Example. In Fig. 1, we demonstrate a simplified online credit card fraud detection task in e-commerce platforms [56], in which a suspect attempts to obtain short-term credit from a credit card via illegal transactions. To achieve this, the suspect (802) makes sham purchases paid by a credit card from a conspired merchant (803). The merchant, after receiving the money from the bank (101), transfers the money through a series of middlemen (804, . . . , 807) with other fraudulent transactions back to the suspect (802). In this scenario, the OLTP system maintains four tables (PERSON, TRANSACTION, CLIENT, and BANK), from which one can create a graph showing relationships of transactions among normal users, suspects, merchants, and middlemen. Whenever a new tuple occurs in the TRANSACTION table, the graph should be updated correspondingly. As soon as an upcoming transaction generates a cycle in the graph, an alarm should be triggered instantly to block the transaction for fur-

ther investigation. Therefore, an underlying detection system for such frauds should meet two key requirements simultaneously.

Performance. The *performance degradation* of both relational transaction and graph analytical workloads should be minimal compared to running them separately on specific systems. Thereby, both the transaction and detection should be completed before the user perceives any lag.

Freshness. The *time gap* between transactions committed on OLTP systems and their accessibility on detection systems should be minimal to prevent fraud on time. Recent studies [10, 56] show extreme requirements of 20-millisecond freshness for fraud detection or system monitoring.

In response, several solutions have been proposed to support such workloads. Unfortunately, none of them can simultaneously meet the requirements, as shown in Fig. 1.

Solution ①: graph processing on offline data. To achieve better GAP performance, this solution utilizes existing graph-specific systems (particularly for *static* graphs in many cases) [22, 31, 67, 76, 86], such as GraphScope [29], to handle GAP workloads efficiently. Since data is separately maintained in OLTP and graph-specific systems, an offline data migration with an ETL (Extract-Transform-Load) process is required. However, such a process is often expensive and slow, and results in a high lag between the transactional data in OLTP systems and the extracted graph data in graph-specific systems [82], which deteriorates the *freshness* guarantee.

Solution ②: graph processing on online data. Some OLTP systems [34, 37, 55, 84] attempt to translate graph-related operations into relational operations. However, prior work [21, 73] has found this solution causes *performance* degradation of GAP up to several orders of magnitude due to costly join operations and huge redundant intermediate data. On the other hand, graph databases [6, 7, 27] use a native graph representation to ensure the efficiency of GAP workloads and directly commit transactions on graphs. However, due to the more complicated management (e.g., maintaining adjacency lists instead of inserting a row) in graph databases to fulfill transactions [87], the *performance* of transactions in graph databases is significantly slower than the relational counterparts, which is also demonstrated in our experimental study (§6.2). Further, legacy business logic was usually designed and implemented on relational OLTP systems, and it is inevitable to process a costly migration to the graph databases.

The tradeoff between performance and freshness is still an open problem for dynamic GAP workloads. Fortunately, *hybrid transactional/analytical processing* (HTAP) is a new trend that processes OLTP and online analytical processing (OLAP) simultaneously in the same system. The state-of-the-art HTAP systems usually leverage a loosely-coupled design to guarantee both performance and freshness [19, 38, 45, 50, 65, 82], which gives an opportunity for dynamic GAP workloads. Analogously, we term dynamic GAP workloads on transactional datasets as *hybrid transactional/graph-analytical processing* (HTGAP).

Our approach. This paper presents GART, an in-memory HTGAP system extended from HTAP systems that can be deployed to bridge an existing relational OLTP system with a graph-specific system for requirements of *performance* and *freshness*. GART performs GAP workloads over the graph-specific system with little *performance* degradation. It reuses transaction logs to replay graph data *online* for *freshness* instead of offline data migration. Unlike the prior HTAP systems with only the relational model, GART also has to support the graph data model for GAP. Therefore, there are two unique goals not encountered by HTAP systems.

First, to adapt to rich workloads flexibly, GART needs to convert relational data to graph data transparently. Thus, some concise yet expressive interfaces should be proposed to the database administrator (DBA) for data conversion from the relational model to the graph model. To fulfill this goal, we propose a collection of *graph extraction interfaces* to define rules of relational-graph mapping in a newly designed component called *RGMapping*. We demonstrate that the interfaces are expressive enough to help GART automatically extract property graphs from relational data sources such as transactions.

Second, to guarantee performance, the dynamic graph storage should support both read and write operations efficiently. Existing *general-purpose* dynamic graph storages [26, 33, 87] support complex updates from transactions but provide sub-optimal GAP performance due to poor data locality and expensive concurrency control. Thus, based on observed characteristics of HTGAP workloads, we propose a new efficient dynamic graph storage for HTGAP with three key components: 1) an *efficient* and *mutable* CSR representation that guarantees the locality of edge scan when updating graph topology data; 2) a *coarse-grained* MVCC scheme that reduces the temporal and spatial overhead of versioning; 3) a *flexible* property storage that allows running different GAP workloads on snapshots generated based on their access patterns.

We implement GART by extending VEGITO [65], a state-of-the-art in-memory HTAP system. The extensions include graph extraction interfaces, the dynamic property graph storage, and integrating a unified graph computation system (GraphScope [29]). To demonstrate the efficacy of GART, we have conducted a set of experiments using two popular benchmarks (i.e., LDBC SNB [4] and TPC-C [71]), as well as diverse graph datasets. Our experimental results show that GART outperforms Solution ① and Solution ② by several orders of magnitude in freshness and performance, respectively. Meanwhile, GART outperforms the state-of-the-art general-purpose dynamic graph storage (i.e., LiveGraph [87]) by up to 4.4× for GAP workloads on the LDBC SNB dataset.

Contributions. We extend HTAP systems to support

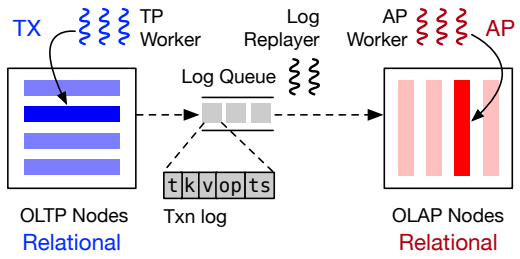


Fig. 2. *The overview of HTAP systems based on a loosely-coupled design.* ***Transaction (Txn) log****: table ID (t), primary key (k), value (v), operation type (op), and timestamp (ts).*

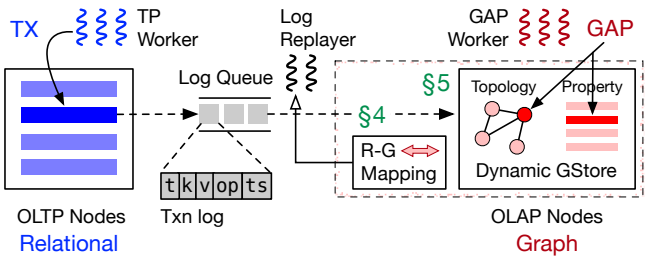


Fig. 3. *The architecture of* GART*. The components in the dashed box are newly designed for HTGAP.*

HTGAP by incorporating relational-graph mapping and a dynamic graph storage into existing computation engines. In summary, the contributions of this paper are:

- The first to extend HTAP architecture for HTGAP workloads with guarantees of performance and freshness (§3) that proposes expressive interfaces of relational-graph mapping for transparent data model conversion (§4).
- A new dynamic graph storage for efficient HTGAP workloads, which is optimized for data locality and concurrency control based on our key insights into HTGAP (§5).
- A prototype implementation (GART) that integrates existing HTAP and GAP systems, as well as a set of evaluations that confirm the efficacy of GART for HTGAP workloads with diverse applications and datasets (§6).

2 Opportunity: HTAP

Hybrid transactional/analytical processing (HTAP) is a new trend that bridges the gap between OLTP and OLAP for real-time analytics on datasets updated by transactions and is already being used in many scenarios [17, 53, 85].

The *loosely-coupled* design is a common choice of state-of-the-art HTAP systems, which dedicate OLTP and OLAP to different physical resources with specific storage types (see Fig. 2). Thereby, it is comparable in *performance* to specific execution engines and can synchronize data with transaction logs to guarantee *freshness* [38, 45, 50, 65, 82]. Specifically, logs of the OLTP node are used to update the extra *column* store on the backup (OLAP node), which is more appropriate for OLAP workloads; the log replayer applies logs in real time on the OLAP node. The log from the OLTP system (*Txn log* in Fig. 2) contains the necessary information for data replaying, such as the identifier of data updates (table ID and primary key), the after-image or delta of the tuple (value), and the temporal meta-data (version number or timestamp) [23, 52, 63, 66, 78]. Reusing logs for HTAP can avoid costly operations for change data capture (CDC).

HTAP systems usually utilize batch-based log replaying for freshness and consistency on the OLAP storage [45, 50, 65]. An efficient design [65] divides time into consecutive and non-overlapping *epochs*. The epoch is automatically increased in a fixed interval, such as several milliseconds, that can trade off between freshness and performance. During each epoch, logs can be replayed in parallel. When an epoch ends, it guarantees all the logs within the epoch have been replayed. Each epoch is specified by an epoch number, which is incremented when entering a new epoch. The system maintains a write epoch number (i.e., wepoch) to represent the epoch when the logs are being replayed. The latest stable epoch number that OLAP users can read is latest_repoch.

Opportunity. We observe that the guarantees of HTAP systems on both performance and freshness can benefit the dynamic GAP workloads. HTAP systems can be extended to process transactions and GAP workloads simultaneously on different storages. Specifically, for **performance**, HTAP systems provide specific storages and execution engines for hybrid workloads, which can fully reuse the effort on specific systems for OLTP and GAP. For **freshness**, logs in HTAP systems contain the updates of relational data, which can be synchronously applied to the graph data in real time without costly operations such as bulk loading, CDC and ETL.

However, to the best of our knowledge, none of the HTAP systems enable native GAP workloads on a dynamic graph storage. To achieve this, HTAP systems need to support conversion between the relational model and graph data model and an efficient dynamic graph storage for HTGAP.

3 Overview of GART

Inspired by the loosely-coupled design of HTAP systems [38, 65, 82], we propose GART, an in-memory HTGAP system that extends HTAP systems by retrofitting the log replayer and the storage for GAP workloads, as shown in Fig. 3. It should be noted that GART can reuse the execution engines of existing OLTP and graph-specific systems.

Architecture and workflow. In GART, transactions are committed in the OLTP nodes and generate logs, like prior HTAP systems [38, 65, 82]. To support rich workloads flexibly and efficiently, GART conducts data model conversion. Relational data in logs need to be converted to graph data and stored in a dynamic graph storage (GStore) of OLAP nodes. GART allows the DBA, who is responsible for defining the database schema, to define the relational-graph mapping through the *RGMapping* component. RGMapping guides the *log replayer* to convert the relational data in logs to the updates on graph data.

GART devises a new *dynamic graph storage* for real-time graph updates and different GAP workloads. Graph data consists of a topology and properties. The topology contains vertices and edges (i.e., an ordered pair of vertices), and the properties are a set of attributes for each vertex or edge. The storage always provides consistent snapshots of graph data (identified by an *epoch*) derived from relational data. Similar to HTAP systems (§2), the log replayer updates the storage with the epoch number wepoch. GAP workloads can read a fresh snapshot or earlier using an epoch number that does not exceed latest_repoch.

GART follows a loosely-coupled design, which can be deployed as a single-machine system or a distributed system that separates OLTP and GAP components (the dashed box in Fig. 3) on different machines for better performance isolation. For the distributed deployment, a crash of the GAP component will not stall the execution of the OLTP component. The graph data can be recovered from the relational data according to persistent RGMapping data. When the OLTP component fails, the existing fault-tolerance mechanism in HTAP systems still works [65], which is orthogonal to our work. In addition, we focus on in-memory processing that can buffer hot data in real-time GAP tasks and meet the freshness and performance requirements. GART is independent of whether the OLTP system is in-memory or not.

To support HTGAP workloads, GART should fulfill two unique design goals never encountered in prior work.

Goal 1: Transparent data model conversion (§4). In HTAP systems, the conversion does not change the data model and only depends on the schema of relational data (e.g., from row store to column store [38, 65]). However, the conversion between different data models for HTGAP workloads requires more semantic information. For example, it needs the mapping between relational tables and vertex/edge types, and the mapping between relational attributes and vertex/edge properties. Prior work [37, 55, 72] uses interface extension rather than data conversion, such as graph extensions on relational databases, which demands users to manually rewrite transactions or change log formats.

Goal 2: Efficient dynamic graph storage (§5). For the HTGAP system, write operations (from the log replayer) and read operations (from the GAP worker) are executed concurrently on the graph storage. The performance of both is important. Although many general-purpose dynamic graph storage systems [30, 32, 54, 87] have existed, their read performance for GAP workloads is sub-optimal due to neglect of HTGAP characteristics. The locality of read operations is sacrificed to guarantee the write performance and transaction semantics. First, adjacency-list-based topology storages ignore the locality of edge scan. Second, fine-grained versioning is expensive and breaks both the spatial and temporal locality. Third, property storages based on a column store cannot guarantee the locality of access patterns among different GAP workloads.

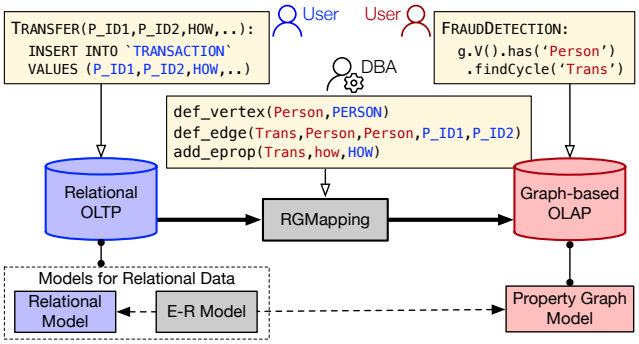


Fig. 4. *An example of data manipulation and graph extraction interfaces provided by* GART *using the dataset in Fig. 1.*

4 RELATIONAL-GRAPH MAPPING

To convert relational data to graph data automatically, it is necessary to provide a relational-graph mapping mechanism. GART uses the *property graph model* and provides the interfaces with the intuition from the *entity-relationship (E-R) model*. The property graph model [11] has been widely adopted to model graph-structured data. As shown in the lower left corner of Fig. 1, a property graph model defines a directed graph topology in which a vertex represents an entity and an edge from a source vertex to a target vertex represents a relationship. Each vertex (resp. edge) belongs to a vertex (resp. edge) type and has a property with attributes (Attr-Value pairs).

4.1 System Interfaces

The conversion from relational data to graph data needs additional semantic information. An intuitive solution is to directly add graph information to the transactions, such as graph extensions for relational databases [48, 55, 69], so that additional information can be added to the log. However, this solution has to extend the interface of the OLTP engine and change the log format; it implies that transactions must also be rewritten manually. Instead, GART decouples the interface into two groups, which are exposed to the user and the DBA, as shown in Fig. 4.

Data manipulation interfaces. GART integrates specific execution engines for OLTP and GAP workloads and retains their user interfaces. Therefore, existing transactions and graph queries can run directly on GART. As the example in Fig. 4 shows, users can execute a transaction called TRANSFER to transfer money. Meanwhile, users can run a query called FRAUDDETECTION to find all cycles on the graph consisting of Person vertices and Trans edges.

Graph extraction interfaces. The interfaces of the *RGMapping* component define the relational-graph mapping, which guides the log replayer to perform data conversion automatically. Fig. 5 lists two kinds of graph extraction interfaces.

Interfaces for adding vertices. In GART, each vertex type corresponds to one table in the relational model, and each

```
# Definition for vertices
def_vertex(vtype,table)
add_vprop(vtype,vprop,attr)
# Definition for edges
def_edge(etype,src_vtype,dst_vtype,pk) # 1-to-m
def_edge(etype,src_vtype,dst_vtype,src_pk,dst_pk) # m-to-m
add_eprop(etype,eprop,attr)
```

Fig. 5. *The graph extraction interfaces provided by* GART. *Arguments from the graph model and the relational model are shown in red and blue, respectively.*

property of vertices corresponds to one attribute in the table. The interfaces def_vertex and def_vprop are used to add new entities and construct the corresponding vertices. Specifically, def_vertex defines a type of vertices (vtype) according to the corresponding table (table). The attributes (attr) of the table can be further mapped to the properties (vprop) of vertices through the interface add_vprop.

Interfaces for adding edges. A relationship in the relational model can be added as a directed edge through the interface def_edge, where etype, src_vtype and dst_vtype correspond to the edge type, the type of source and destination vertices of this type of edges, respectively. To distinguish between different relationship types, RGMapping provides two def_edge for 1-to-m relationships (also 1-to-1 relationships) and m-to-m relationships, respectively. The difference between them lies in whether the interface requires the primary keys (pk) of one table or both tables as inputs. Furthermore, add_eprop is used to add the edge property (eprop).

The graph extraction interfaces make the OLTP engine and log formats unchanged. Moreover, unlike data manipulation interfaces, graph extraction interfaces are used only when defining the graph schema instead of used in each request. DBAs can define the RGMapping for a fixed data model just once according to workloads. For complex data models, DBAs can use automatic E-R model generation tools [1, 5] as a guide according to the relational schema, even if they have less knowledge about the workloads.

4.2 Expressiveness of RGMapping

We next show that graph extraction interfaces are expressive enough to map relational data to a property graph modeled by the same E-R model.

The E-R model has shown its powerful expressive capability in describing relationships and has been widely adopted in defining relational data [20, 28]. Generally speaking, the E-R model contains a set of *entities* with *attributes*, which usually represent objects in the physical world, and describes the *relationships* between entities. Intuitively, *for any E-R model, there exists a unique property graph schema that represents the E-R model* [59]. Entities and relationships can be mapped to vertex types and edge types of the property graph model, respectively, and use properties to store attributes. The interfaces also help DBAs extract a subgraph from the property graph. Note that edges in graphs are binary (2-ary) relations. An n-ary relationship of order greater than two can

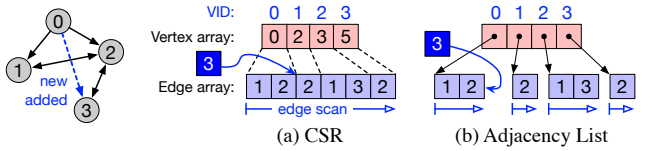


Fig. 6. *Two typical representations of an example dynamic graph topology, namely (a) CSR and (b) adjacency list.*

be mapped as a type of vertices with n associated edges.

Back to the example in Fig. 1 and Fig. 4, assume that a mapping scheme between the E-R model and the relational model has been defined (lower left dashed rectangles in Fig. 4). There is an entity called Person and a relationship called Trans that exists between instances of the Person entity in the E-R model. The DBA can utilize the interfaces (the middle part in Fig. 4) to define a property graph model that contains one type of vertices (Person) and one type of edges (Trans), which are derived by the entity Person and relationship Trans, respectively. Meanwhile, there is a property (How) on Trans derived from the HOW attribute.

RGMapping can map changes to relational data to property graphs *on-the-fly*. Depending on whether the data involves entity tables or relationship tables, the log replayer converts them to vertices or edges, respectively. Users can customize the extracted graphs partially so that the graphs extracted do not have to exactly match the E-R model.

5 DYNAMIC GRAPH STORAGE

The graph storage of GART stores the graph topology and properties and provides two kinds of operations: read for GAP workloads (e.g., edge scan) and write for the log replayer (e.g., insert and delete). For the graph topology, compressed sparse row (CSR), a compact graph representation, is widely adopted by (static) graph systems [29, 43, 73, 77], as shown in Fig. 6(a). However, CSR is also notoriously inefficient for dynamic workloads on the graph (e.g., edge insertion and deletions). Therefore, dynamic graph storage systems [30, 32, 33, 41, 54, 87] commonly use adjacency lists (based on linked lists or vectors) to store the graph topology (see Fig. 6(b)). For vertex (resp. edge) properties, a columnar storage is usually employed to efficiently read the same property for all vertices (resp. edges) with the same type [29, 55]. In addition, the dynamic graph storage also needs to record the version of vertex/edge/property updates for MVCC [30, 87].

The above traditional design has several performance issues for HTGAP workloads. First, using adjacency lists suffers from poor locality when sequentially scanning edges of all vertices, which is a common yet costly operation in GAP. A cache miss may occur when scanning the edges of an adjacent vertex. For example, transactions may insert edges randomly, resulting in unordered memory allocation for new edges of different vertices. Second, using fine-grained versioning for each vertex or edge update imposes a significant performance penalty for GAP workloads, since checking the

version for each read operation breaks both spatial and temporal locality and introduces additional overhead. Third, the existing property storages cannot guarantee locality flexibly for different access patterns among GAP workloads.

Key insights. Some unique characteristics of HTGAP workloads open opportunities to exploit the locality of a dynamic graph storage. Note that we term the time gap as the interval between a transaction committing an update and a graph query reading it. First, *the time gap in HTGAP (typically a dozen milliseconds, which is equal to the freshness of* GART *shown in Table 2) is sufficient to update a compact structure like CSR*. Thus, GART can still use a CSR-like storage for the graph topology instead of adjacency lists to improve the locality of edge scan. Second, *the GAP latency is almost always much longer than the time gap*. It implies that assigning versions to each update (fine-grained MVCC) is not necessary for GAP workloads. Thus, GART can use coarse-grained MVCC (i.e., at epoch granularity) to reduce memory and computation overhead, even though the committed updates cannot be read immediately. Third, *the access pattern of each GAP workload is usually fixed and easily detectable*. Given an HTGAP workload, fixed correlations between different attributes can be found by parsing the requests. For example, some attributes (e.g., balance and payment) may always be updated or read together. Therefore, GART can allow users to decide how to store different properties.

General idea. Based on the insights, we devise a new dynamic graph storage for HTGAP workloads. Fig. 7 illustrates the main structure of the dynamic graph storage for one type of vertex and edge. For the graph topology, a variant of CSR is proposed to exploit locality of edge scan, where the edge array is divided into multiple *edge segments* for dynamic updates. Using edge segments offers a tradeoff between read and write performance and allows the structure to be updated in batches. The vertex array is indexed by vertex ID (VID) and contains the links to the edges (neighbors) of each vertex. To reduce the overhead of edge insertion, the edges of each vertex are further divided into multiple *edge blocks*. Each vertex stores a pointer (tail) to the last edge block in the vertex array (not shown in Fig. 7 due to space constraints). To enable MVCC at epoch granularity, each vertex maintains an *epoch table*, which links to the edges inserted in the same epoch. It avoids attaching versions to each edge as in fine-grained MVCC. Similar to CSR, the epoch table stores the logical offset of the first edge for each (read) epoch number. In addition, vertex (resp. edge) properties are stored in *property blocks* within the vertex array (resp. edge segment). Each property block is a column store for one property or a group of correlative properties (column-family), which is indexed by the corresponding vertex (resp. edge) offset in the vertex array (resp. edge segment). Finally, a new interface is provided for combining correlative properties into a column family following access patterns of HTGAP workloads.

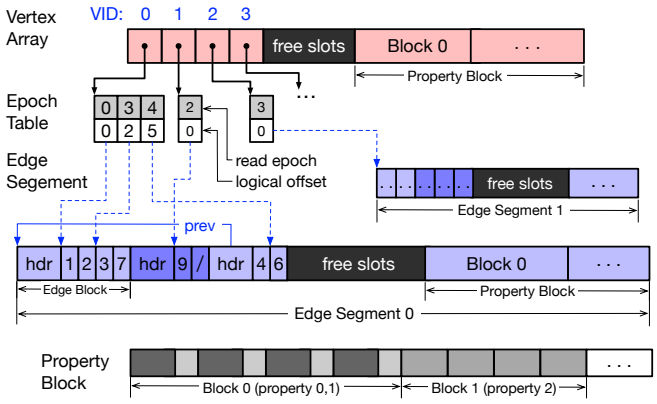


Fig. 7. *The key structure of the dynamic graph storage in* GART *for one type of vertex and edge.*

5.1 Efficient and Mutable CSR

GART devises an efficient and mutable CSR that guarantees high performance in both scans and updates on the graph topology, providing data locality similar to an immutable CSR. Each edge segment has a fixed initial size (e.g., 16KB) and stores edges (i.e., neighboring vertex IDs) of a fixed number of vertices (e.g., 4,096). The free slots are reserved for new edge blocks. Each vertex has a group of *edge blocks*, and new edges will be inserted into the tail edge block.

Fig. 7 shows an example where each segment stores the edges of two vertices (e.g., vertex 0 and 1). Initially, edges of the same vertex are stored consecutively in an edge block, so there is no overlap between edges of different vertices in an edge segment. As edges are continuously inserted, a vertex will allocate new edge blocks, which form a linked list (e.g., vertex 0). Each edge block has a header block (hdr) to store the meta-data of edges, such as the block size, the number of valid edges, and the pointer (prev) to the previous edge block of the same vertex.

Edge scan. Given a read epoch number, GART first uses the tail pointer and the epoch table of the vertex to find edges of that epoch within its edge blocks, and then scans edges forward based on the prev pointer stored in the header block (hdr). Note that each edge block only needs to be addressed once, which has little performance impact. In the beginning, GART can provide data locality comparable to vanilla CSR. However, after inserting numerous edges for different vertices, the edge blocks of a vertex will form a long linked list, leading to performance degradation in edge scan. To mitigate this issue, GART compacts an edge segment periodically or when the segment is close to full. After compaction, all edge blocks of the same vertex will be merged into one edge block (e.g., the first edge block of vertex 0).

Insertion. For a new vertex, GART atomically inserts it into a free slot of the vertex array and initializes an empty epoch table for it. When inserting an edge (e.g., from vertex 1 to vertex 4), the destination vertex ID will be directly inserted

into the tail edge block of the source vertex, if the edge block is not full (e.g., vertex 1 in Fig. 7). Otherwise, a new tail edge block of double the size is first allocated from the free slot of the edge segment, and then the edge is inserted into it. To reduce fragmentation, when the size of the tail edge block is smaller than a threshold, all edges will be moved to the newly allocated edge block to further improve data locality; the original edge block will be skipped. When an edge segment is full, a new segment of double the size is allocated, and all edges in the original segment are moved to the new one.

The write conflicts when concurrently inserting edges to the same vertex are resolved by per-vertex locks. Moreover, to resolve the conflicts when allocating edge blocks for different vertices on a full segment, the log replayer should lock the segment after checking for free space. Specifically, if the segment still has free space, the segment is locked in a shared manner; if the segment is full, the log replayer should exclusively lock the segment first and then allocates a new one.

Deletion. When deleting a vertex (resp. edge), a delete flag is appended to the vertex array (resp. edge block), which records the offset of the deleted vertex (resp. edge). When encountering the delete flag, the deleted vertex (resp. edge) will be skipped during scanning the graph topology. Furthermore, garbage collection (GC) will physically delete the vertices and edges and free up space in the background.

Discussion: structure parameters. We have tuned parameters including: (1) the number of vertices managed by each segment, (2) the initial size of edge blocks and segments, and (3) the resize factor of edge blocks (or segments) when they are full. Increasing (1) results in higher latencies for inserts, but it improves read performance. To balance read and write performance, we set (1) to 4096. The default values of (2) and (3) have minimal impact on read performance. We have adjusted these values to minimize the allocation of edge segments and optimize memory usage.

5.2 Coarse-grained MVCC

GART employs a *coarse-grained* MVCC scheme to reduce the temporal and spatial overhead of fine-grained MVCC. The scheme is based on the key observation that GAP workloads usually run longer at low concurrency than transactions. Thus, GART can enable MVCC at epoch granularity. In particular, the edge storage needs to adapt to the epoch instead of a fine-grained version for each edge. For each vertex, the epoch number of edges increases with the logical offsets. Since edges are append-only in edge blocks, edges with the same epoch number are consecutive. Therefore, GART can use an epoch number for a batch of edges.

Specifically, each vertex maintains an *epoch table* to store the offset in the edge segment for each epoch. As the example shown in Fig. 7, the 3rd to 5th edges of vertex 0 (offsets 2–4) are all inserted at epoch 3. At epoch 4, the GAP worker

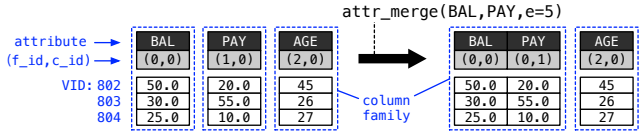


Fig. 8. *An example of the flexible property storage.*

will read all edges with the logical offset less than 5 at vertex 0. Since the logical offset of each epoch is immutable, GART can scan edges sequentially as if on a static graph. This design avoids checking versions for each edge and maintains the compact edge storage like in CSR.

When the log replayer inserts an edge with the epoch number, it will check whether the epoch number exists in the epoch table of the source vertex. The transaction protocol guarantees that the new epoch number must be greater than all existing epoch numbers [65]. If the epoch number does not exist, a new epoch number and the offset will be appended to the epoch table atomically.

The epoch table is stored as a ring buffer, since older snapshots are more likely to never be read again. For the corner cases where the oldest epoch number is still in use, GART uses a classical allocation amortization technique, similar to the C++ STL vector, to extend the ring buffer.

5.3 Flexible Property Storage

The storage model of properties influences the performance of read operations over properties. However, there exists no efficient property storage model for all GAP workloads. To support different GAP workloads in a more efficient way, GART presents a *flexible property storage* that allows system users to define the storage model according to application memory access patterns. Initially, GART utilizes a column store to store property data, which stores the same property (attribute) of the same type of vertices/edges continuously and is friendly to workloads that scan one or several attributes sequentially and independently.

To improve the property scan performance under different scenarios, users can combine several high-related attributes into a *column family* with the attr_merge interface *ahead of time*, as shown in Fig. 8. The Person vertex consists of three attributes: Bal, Pay, and Age. The meta-data of each attribute maintains the column family index (f_id) and the column index in the column family (c_id). The initial f_ids are different due to the pure column store.

As time goes on, the access patterns to some attributes may have certain correlations. Users can use the attr_merge interface to merge some attributes into the same column family *on-the-fly*, and generate a snapshot with a read epoch number. For example, if the attributes Bal and Pay are accessed together in a long-term GAP workload, users can merge them into a column family with read epoch 5. GART needs to generate a new version of the meta-data of these two attributes with epoch number 5, and copy them into a column family in the background. Then, the workload is processed on the new

Table 1: *Datasets used in evaluation, including TPC-C (warehouse=20) [71]* (CH)*, SNB-SF-10 [4]* (SB)*, Wiki [36]* (WK)*, R-MAT [18]* (RM)*, UK-2005 [16]* (UK)*, and Twitter-2010 [42]* (TT)*.*

Graphs	CH	SB	WK	RM	UK	TT
$\|V\|$	700 K	7.5 M	5.7 M	5.0 M	39.5 M	41.7 M
$\|E\|$	6.0 M	8.8 M	130 M	300 M	936 M	1.47 B

snapshot, which can also avoid some pre-processing tasks (e.g., the projection phase of GNN workloads to obtain required properties). We leave the automatic plan generation and attribute merging as future work.

6 Evaluation

We have implemented GART by extending VEGITO [65], a state-of-the-art in-memory HTAP system. VEGITO adopts DrTM+H [79] as the OLTP engine and supports tens-of-millisecond freshness with millions of transactions per second. This makes the HTGAP system design more challenging than in the case of low OLTP throughput. The extensions include the log replayer with relational-graph mapping and a new dynamic graph storage. We further integrated an open-sourced one-stop graph processing engine GraphScope [29] into GART, in order to support diverse GAP workloads.

6.1 Experimental Setup

Testbed. All evaluations are conducted on two dual-socket machines. Each machine has two 12-core Intel Xeon E5-2650 CPUs, 256 GB DRAM, and two 56 Gbps InfiniBand (IB) NICs via PCIe 3.0 connected to a Mellanox 40 Gbps IB Switch. Since the latency of GAP tasks is much higher than that of OLTP tasks, we end the execution of OLTP tasks after several rounds of GAP tasks to ensure the time of OLTP and GAP is similar in HTGAP workloads. Unless otherwise noted, we dedicate one machine for OLTP requests (OLTP server) and the other for GAP requests (GAP server). On the OLTP server, we pin 20 cores for OLTP worker threads and 1 core for the OLTP client thread. On the GAP server, we pin 12 cores for GAP worker threads, 10 cores for log replayer threads, and 1 core for the GAP client thread. The single core for clients is sufficient for request generation. We set the epoch interval to 15 milliseconds. For edge segment compaction (§5.1), we choose a strategy of compaction at edge insertion instead of periodic compaction, which can reduce repeated compaction when the OLTP throughput is high.

Benchmarks. Considering that there is no standard HTGAP benchmark, we first retrofit LDBC Social Network Benchmark (SNB) [4] and TPC-C [71] as two new HTGAP benchmarks and further select several typical graph datasets as our micro-benchmarks. The graphs are summarized in Table 1.

LDBC SNB is a GAP benchmark that contains a social network graph and different types of GAP workloads. We select to load 50% of edges and insert another 50% of edges for transactions. We set the scale factor (SF) of the dataset to 10 (about 8.4 GB) on each server.

TPC-C is a standard OLTP benchmark that contains relational datasets and five kinds of transactions. We extract two bipartite graphs (ORDER-ORDERLINE and CUSTOMER-ITEM graphs) from relational tables by mapping rows in entity tables as vertices and adding edges based on relationship tables. We deploy 20 warehouses on each server.

On the graphs of LDBC SNB and TPC-C, we choose three types of GAP workloads as prior work [29]:

- Graph analytics (GA): three representative graph algorithms from LDBC Graphalytics Benchmark [3], including PageRank (PR), Connected Components (CC), and Single Source Shortest Path (SSSP).
- Graph traversal (GT) [81]: three scan-dominated queries from LDBC SNB [4], including one interactive query (IS-3) and two business intelligence (BI) queries (BI-2 and BI-3). These queries access both the graph topology and properties.
- Graph neural network (GNN) [75, 83]: inference on three popular models, including Graph Convolution Network (GCN) [40], GraphSage (GSG) [35], and Simple Graph Convolution (SGC) [62].[1]

Comparing targets. To show the efficacy of the loosely-coupled design for HTGAP, we mainly focus on the performance of OLTP and GAP, and the freshness in GART against two different solutions (see §1). For Solution ①, we connect DrTM+H with GraphScope [29] (DH+GS), where transactions are served by DrTM+H (the same OLTP engine of GART), and transactional data are periodically loaded into GraphScope, a state-of-the-art GAP engine. For Solution ②, we evaluate Neo4j [6], a popular graph database that supports both transactions and GAP.[2]

To study the efficiency of our dynamic graph storage, we integrated LiveGraph [87], a state-of-the-art transactional graph storage, into GART (G/LG). LiveGraph uses a highly optimized adjacency list format (similar to Fig. 6(b)) to store the graph topology and a row store with fine-grained MVCC to store properties. GART's graph storage and LiveGraph provide the same interfaces. Therefore, GART outperforms G/LG solely due to three design choices of our graph storage. Note that all systems, except Neo4j, use the same OLTP and GAP engines, namely DrTM+H and GraphScope, with the same configurations (e.g., the number of worker threads) for fairness. Although we try our best to run HTGAP workloads on Neo4j and DH+GS, they still cannot support TPC-

[1] We run graph analytics and graph neural network workloads directly on the CUSTOMER-ITEM graph in the TPC-C dataset and PERSON-POST graph in the LDBC SNB dataset. Graph traversal queries from LDBC SNB are tightly coupled with its dataset. We rewrite queries on the TPC-C dataset and ensure that they have the same computation patterns as LDBC SNB.

[2] We also evaluated TigerGraph [24] as an alternative solution. However, TigerGraph's timestamp only provides second-level accuracy, which limits its ability to evaluate freshness with sub-second precision.

Table 2: *A comparison of OLTP throughput (in transactions per second), GAP latency (in milliseconds), and freshness (in milliseconds) using different workloads among* GART*, a combination of* DrTM+H *and* GraphScope *for offline data processing* (DH+GS)*,* Neo4j *(not support GNN workloads), and* GART *w/* LiveGraph (G/LG)*. Note that ↓ (resp. ↑) indicates low (resp. high) is better.*

Workloads		LDBC SNB				TPC-C	
		GART	DH+GS	Neo4j	G/LG	GART	G/LG
OLTP ↑		1837 K	1929 K	3.5 K	1836 K	245 K	212 K
GA ↓	PR	377	309	5323	1276	204	329
	CC	362	312	4726	1137	210	300
	SSSP	513	433	4668	1381	315	410
GT ↓	IS-3	17.9	16.9	2.0	18.0	14.2	14.6
	BI-2	235	201	568	828	1884	2806
	BI-3	292	266	573	1278	266	586
GNN ↓	GCN	1097	940	×	1834	623	636
	GSG	1774	1443	×	2502	386	418
	SGC	779	717	×	1237	184	257
Freshness ↓		18	15683	5	25	18	25

C due to costly code transcription and graph extraction from complex logs, respectively.

Coding effort. To extract graph data from relational data, we only write about 10 LoCs for each benchmark (LDBC SNB and TPC-C), thanks to graph extraction interfaces in GART (§4.1). This is far less code than OLTP and GAP programs, which can be inherited directly from existing specific systems. In contrast, we write 584 LoCs in DH+GS for loading graph data and 70 LoCs in Neo4j for rewriting transactions.

6.2 Overall Performance

We first show the overall performance of all baselines for HTGAP workloads in Table 2. We use transaction throughput as the evaluation metric for OLTP workloads, and computation latency (execution time) for GAP workloads. We also evaluate the freshness of each system, namely, the maximum time delay between an update was committed in OLTP and this update is visible in GAP workloads [65]. In a nutshell, among all baselines, only GART can simultaneously satisfy the requirements of *performance* and *freshness*.

OLTP performance. For systems based on the loosely-coupled design (GART, G/LG) and DH+GS, OLTP throughput is not impacted by GAP workloads. The peak throughput of GART can reach over 1,837,000 and 245,000 transactions per second on datasets LDBC SNB and TPC-C, respectively. GART performs 2-3 orders of magnitude better than Neo4j (Solution ②). Neo4j has the lowest OLTP performance as the graph data model is less efficient than the relation model for OLTP workloads. GART and G/LG show an OLTP throughput reduction of only 5% compared to DrTM+H with offline data processing (DH+GS). This result demonstrates that the loosely-coupled design can support HTGAP without necessarily sacrificing OLTP performance.

GAP performance. Among all baselines, DH+GS achieves the best GAP performance, as its graph data is stored as a *static* graph, which uses compact graph representation without concurrency control (e.g., MVCC). However, its freshness is extremely high. Neo4j performs much worse than GART and G/LG due to its adjacency-list-based storage [68, 87]. GART greatly outperforms G/LG except for the IS-3 query, thanks to our dynamic graph storage which takes the characteristic of HTGAP workloads into consideration (breakdown details in §6.3). The IS-3 query only involves a very limited graph data size, thus the overall execution time is dominated by cross-language invocation overheads. The backend and frontend engines of GraphScope are developed with Rust and C++, respectively.

Freshness. The freshness of GART is about 18 ms, which is much lower than most GAP latencies and is independent of datasets. It is three orders of magnitude better than DH+GS (Solution ①). Similar to VEGITO [65], the freshness of GART is only determined by the epoch interval (15 ms). DH+GS has the worst freshness (more than 15 seconds), which is unaffected by the ETL frequency and depends on the graph data size. This is because the immutable graph storage requires the entire graph to be reloaded whenever changes are made. The freshness of G/LG is 25 ms due to the lower write performance of the graph storage. The freshness of Neo4j is only about 5 ms, as data is committed in place.

6.3 Breakdown Analysis on GAP Performance

From Table 2, we observe that GART achieves a large performance improvement over other systems on graph analytics and traversal workloads and a relatively small improvement on GNN workloads. To gain a deeper understanding of the dynamic graph storage in GART, we perform a detailed comparison with G/LG.

We split a single execution of a GAP query into three parts: (P1) accessing the graph topology; (P2) getting properties from visited vertices or edges; and (P3) computation over the graph topology and properties obtained from (P1) and (P2) parts. Meanwhile, the performance gain of GART mainly comes from three aspects: (A1) the efficient and mutable CSR with good locality of edge scan; (A2) the coarse-grained MVCC that alleviates the costly versioning; and (A3) the flexible property storage that adapts to access patterns. Table 3 shows the results of a breakdown analysis of three GAP workloads for the LDBC SNB dataset. Since G/LG and GART have the same backend GAP engine, their performance in part (P3) is nearly identical. Meanwhile, the design differences between (A1) and (A2) affect the performance of the (P1) part, while the performance of the (P2) part is influenced by (A3).

Graph analytics. Table 2 shows the execution time of dif-

Table 3: *Breakdown analysis of G/LG and GART over three GAP workloads (in milliseconds) using the LDBC dataset.*

	Storage	Topo (S1)	Prop (S2)	Comp (S3)	Total
PR	GART	107 (28%)	163 (44%)	107 (28%)	377
	G/LG	658 (52%)	486 (38%)	132 (10%)	1276
BI-3	GART	10 (3%)	178 (61%)	104 (36%)	292
	G/LG	40 (3%)	1115 (87%)	123 (10%)	1278
SGC	GART	36 (5%)	477 (61%)	266 (34%)	779
	G/LG	236 (19%)	732 (59%)	269 (22%)	1237

ferent algorithms and excludes the time of storing outputs. The latency of PageRank, CC, and SSSP on G/LG is 2.5×, 2.3×, and 2.0× higher than GART, respectively. The performance gain of GART on graph analytics workloads is mainly from the better performance of accessing the dynamic graph topology due to (A1) and (A2), and the higher efficiency of obtaining required properties thanks to (A3). For example, as shown in Table 3, the end-to-end execution time of PageRank is dominated by the graph traversal (edge scan) operations (P1) and getting required properties (P2). GART's graph storage outperforms G/LG by 6.2× in (P1), with 82% of the improvement attributed to (A1). GART also outperforms G/LG by 3.0× in (P2) due to (A3). Note that PageRank uses only one property, so GART does not group it with others.

Graph traversal. For graph traversal workloads, GART's latency is 3.5× and 4.4× lower than G/LG's for two BI queries (BI-2 and BI-3) on the LDBC SNB dataset, while GART performs almost as well as its competitor on the interactive query (IS-3). The reason is that compared with BI queries, the interactive query only involves a very limited graph data size (vertices and edges as well as their properties), and our optimized storage design cannot contribute much in such a circumstance. Instead, the computations of two BI queries rely on scanning one or several properties of a large number of vertices or edges, and (A1) to (A3) can benefit a lot. Observe that given a BI query (BI-3), compared with G/LG, GART performs 4.0× and 6.3× faster on accessing the graph topology (P1) and obtaining required properties (P2) parts, respectively (see Table 3).

Graph neural networks. On three GNN workloads, GART outperforms G/LG by 1.3× on average. GNN workloads need several/all properties of vertices/edges as raw features to conduct GNN computation, and (A3) allows users to store and get required properties in a more efficient way. As we can see from Table 3, on SGC, GART is 1.5× faster than G/LG in obtaining the required properties in (P2). Meanwhile, (A1) and (A2) make GART 6.6× faster than G/LG in (P1). The performance improvements over the TPC-C dataset are relatively small, because the CUSTOMER-ITEM graph in dataset TPC-C is very dense, and the end-to-end execution time of GNN workloads is dominated by the complex computations over properties of vertices or edges (P3).

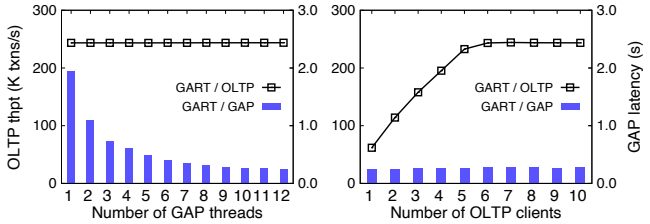


Fig. 9. *Performance isolation on GART with the increase of (a) GAP workloads and (b) OLTP clients, respectively.*

6.4 Performance Isolation

To demonstrate the performance isolation in the HTGAP workloads, we evaluate the OLTP and GAP performance with the increase of GAP and OLTP clients. We use the TPC-C benchmark as the OLTP workloads and execute PageRank on the CUSTOMER-ITEM graph derived from the TPC-C schema. We use the number of OLTP clients and the number of worker threads for a single GAP request to control the workloads.

In general, GART provides strong performance isolation between OLTP and GAP workloads. Fig. 9(a) shows the performance of OLTP and GAP workloads when we gradually increase the number of GAP worker threads. The OLTP performance degradation is trivial (1%), even if the GAP workloads are saturated. This is due to the physical isolation in GART, as GAP workloads do not interfere with transactions. On the other hand, when we increase the number of OLTP clients, as shown in Fig. 9(b), the performance degradation of GAP workloads is about 12%. This is because the number of edge versions also increases, causing additional overhead to check versions. At 5 clients, the OLTP server's maximum capacity is reached due to a fixed number of cores being allocated for its use.

6.5 Graph Topology Storage

Edge scan is a common and costly operation in GAP workloads, such as PageRank and LPA. To study how the performance of edge scan is affected by different graph topology storages, we compare the following typical graph storages.

- **CSR** is a compact structure without the support of updates, which is widely used by the *static* graph topology storages.
- **LiveGraph** [87] is a state-of-the-art dynamic graph storage that uses adjacency lists and fine-grained MVCC.
- **SegCSR** is a CSR-like topology storage proposed by GART, which uses segment-based design and coarse-grained MVCC. Note that each edge segment has a 1 MB initial size and manages 4,096 vertices.
- **SegCSR/TS** is similar to SegCSR, except that it uses fine-grained MVCC as LiveGraph.

To simulate diverse workloads, we load graphs in two patterns: (1) *bulk load*, i.e., edges are sorted by their source vertices and loaded sequentially, and (2) *random insertion*, i.e., edges are loaded randomly to simulate the behavior in trans-

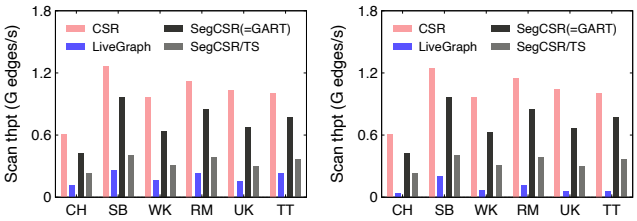


Fig. 10. *Comparison of single-version edge scan perf. for different topology storages using (a) bulk load and (b) random insertion.*

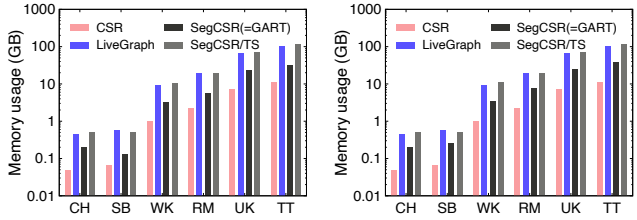


Fig. 11. *Comparison of memory usage (in log scale) for different topology storages using (a) bulk load and (b) random insertion.*

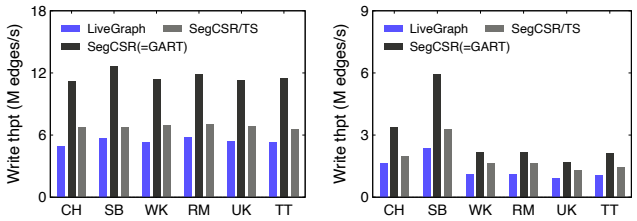


Fig. 12. *Comparison of write throughput for different topology storages using (a) bulk load and (b) random insertion.*

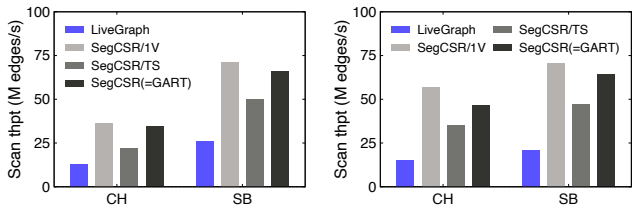


Fig. 13. *Comparison of multi-version edge scan perf. for different topology storages using (a) bulk load and (b) random insertion.*

actions. We scan edges of each vertex and evaluate edges read per second as scan throughput.

Read-only workloads. We first evaluate the performance of the single-version edge scan without version checking. As shown in Fig. 10(a), with the bulk load, SegCSR exhibits consistent performance behavior across various datasets. For example, SegCSR only incurs a 35% slowdown compared to CSR for WK, while LiveGraph incurs more than an 80% slowdown. Compared with adjacency lists for each vertex in LiveGraph, SegCSR has a better locality and fully exploits CPU prefetching as it associates the edges of many vertices in a segment. Moreover, the edge scan throughput of SegCSR is more than 2× that of SegCSR/TS since SegCSR adopts a simpler data structure for edges. With random insertion, the locality of LiveGraph suffers from memory allocation. As shown in Fig. 10(b), SegCSR outperforms LiveGraph by up to 12.5× (from 4.7×) and only incurs about a 30% slowdown compared to CSR.

The memory usage is shown in Fig. 11. Compared with CSR, SegCSR requires about 3× memory of CSR for updates, which is significantly less than that of LiveGraph (8.8× of CSR). The coarse-grained MVCC helps SegCSR reduce memory largely by using the epoch table for each vertex instead of timestamps for each edge. Compared to SegCSR/TS, SegCSR reduces memory usage by up to 3.8×. The memory usage of SegCSR/TS is higher than that of LiveGraph (typically less than 18%) due to the free slots in the edge segments.

Read-write workloads. We further evaluate the performance of the write and the multi-version edge scan. We scan the edges in latest stable version via latest_repoch (§2) when a dedicated number of edges have been inserted. For the bulk load setting, we use the ORDER-ORDERLINE graph in TPC-C and PERSON-POST graph in LDBC SNB. With the random insertion setting, we utilize a shuffled CUSTOMER-ORDER graph in TPC-C and PERSON-POST graph in LDBC SNB.

As shown in Fig. 12, the write throughput of SegCSR is about 2.2× and 2× of LiveGraph with the bulk load and random insertion, respectively. According to the performance of SegCSR/TS, about 70% of the performance improvement is due to the fact that the coarse-grained MVCC of GART writes the newly generated version (epoch) number to the epoch table only once, instead of writing the version number on every update. Moreover, writes of SegCSR do not perform costly GC-related operations, unlike with LiveGraph. SegCSR will copy edges from an old segment with insufficient space to a new segment with a larger space. It introduces high tail latency (more than 7,000× of ordinary edge insertion latency on average) in edge insertion, but the frequency of it being triggered is less than 0.01%.

As shown in Fig. 13(a), with the bulk load setting, read performance of SegCSR outperforms LiveGraph by 2.5× due to better locality and coarse-grained MVCC. It is very close to the upper bound (single-version read performance, SegCSR/1V), while LiveGraph is about 37% of SegCSR/1V. To show the efficiency of coarse-grained MVCC, SegCSR/TS outperforms LiveGraph only by 1.7×. With the random insertion setting shown in Fig. 13(b), SegCSR and SegCSR/TS outperform LiveGraph by 3.1× and 2.3×, respectively. It indicates that the coarse-grained MVCC in GART can largely increase read performance for dynamic workloads.

6.6 Flexible Property Storage

To study the performance of the flexible property storage, we compared it with two typical property storages: row store (row) and column store (col). For the read (resp. write) performance, we scan (resp.update) the properties of each vertex using CUSTOMER vertices derived from TPC-C. We control the number of columns scanned and updated, and evalu-

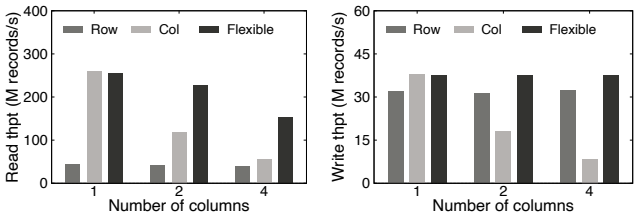


Fig. 14. *Comparison of (a) read throughput and (b) write throughput for different property storages with different numbers of columns read and written, respectively.*

ate the number of processed vertices per second as read and write throughput, respectively.

Fig. 14(a) reports the read throughput. The performance of the row-based property storage is fixed as the number of scanned columns increases since it needs to fetch at least one cache line. However, the performance of column-based property storage significantly drops due to cross-column access. Compared with the row-based storage, the flexible property storage achieves better performance, especially when scanning a few columns. For example, the flexible property storage achieves 5.9× read throughput when scanning only one column. The write operations of the flexible property storage also outperform existing storage models, as shown in Fig. 14(b). It achieves a speedup of 1.2× and 4.4× compared to row-based and column-based storages, respectively, when writing four columns of properties.

Row-based storages and column-based storages have different performance behaviors for reads and writes. We find that the read operation is light and dominated by the memory footprint, while the write operation is dominated by the number of writes due to the overhead of memory copy and atomic operations. The flexible property storage allows users to combine attributes into a column family *on-the-fly* with some overhead. In our experiments, it takes about 1.1 seconds to create a property snapshot with 4 columns as a column family for 229 MB properties.

7 Related Work

HTAP systems. HTAP systems have three main typical design choices. Dual systems [51, 57, 58, 82] combine two specialized systems for OLTP and OLAP scenarios, while single-layout systems [39, 60, 64] support HTAP workloads from either an OLTP or an OLAP system. Dual-layout systems [9, 14, 15, 19, 44, 50] aim to build a single system with different data layouts for the two scenarios. Vegito [65] is proposed to retrofit fault-tolerant backups to support hybrid workloads, which arrives at a sweet spot for the performance-freshness tradeoff. These works are developed for relational data, while GART extends Vegito to constantly maintain a graph layout for transactional data from an OLTP system to support dynamic graph analytical processing.

Graph databases. Graph databases [2, 6, 8, 27] support both OLTP and GAP in a single system. In order to conduct efficient graph updates, they typically adopt linked lists to store adjacency lists, which downgrades the performance of edge scan and the whole GAP workloads. LiveGraph [87] devises the Transactional Edge Logs (TELs) based on adjacency lists to support both efficient sequential scan and edge insertion. Adding a CSR-based in-memory property graph representation in a relational database has been investigated by Oracle, but without support for updates [13]. GART decouples OLTP and GAP execution to make both workloads more efficient.

Dynamic graph systems. Prior work presents many general-purpose dynamic graph systems [24, 30, 41]. Terrace [54] uses PMA [25, 80], a dynamic memory array based on tree-based index structures, to store edges of streaming graphs. CSR++ [32] combines segmented vertex arrays and vector-based adjacency lists for each vertex, which does not guarantee the locality of edge scan from adjacent vertices (see Fig. 6(b)). Moreover, CSR++ does not support multi-versioning. Sortledton [33] provides a general-purpose and transactional graph data structure based on adjacency lists. Teseo [26] and LLAMA [49] are also CSR-like and guarantee the locality of edge scan. While general-purpose dynamic graph storages can replace GART's storage in functions, they may face performance issues. For example, GART could use LLAMA [49] as the graph storage, but it would have to copy data pages for each snapshot, which would be inefficient for scenarios with high data generation rates or extreme freshness. Based on insights from HTGAP, the graph storage of GART does not need to support full transactional semantics, allowing for new designs and optimizations in the topology storage. In addition, GART also provides the flexible property storage for diverse GAP workloads.

8 Conclusion

This paper presents GART, the first hybrid transactional and graph analytical processing (HTGAP) system based on a loosely-coupled design. It proposes expressive interfaces for transparent data conversion and an efficient dynamic graph storage with good locality. Evaluations confirm its efficacy and efficiency. The source code of GART, including all benchmarks, is available at https://github.com/SJTU-IPADS/vegito/tree/gart.

Acknowledgment

We sincerely thank our shepherd Jean-Pierre Lozi and the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (No. 62272291, 61925206), the Fundamental Research Funds for the Central Universities, the HighTech Support Program from Shanghai Committee of Science and Technology (No. 22511106200), and a research grant from Alibaba Group through the Alibaba Innovative Research Program. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

REFERENCES

[1] erwin Data Modeler. https://www.erwin.com/.

[2] JanusGraph. https://janusgraph.org/.

[3] LDBC Graphalytics. https://ldbcouncil.org/benchmarks/graphalytics/.

[4] LDBC Social Network Benchmark (LDBC-SNB). https://ldbcouncil.org/benchmarks/snb/.

[5] Navicat. https://navicat.com/.

[6] Neo4j. https://neo4j.com/.

[7] Neptune. https://aws.amazon.com/neptune/.

[8] OrientDB. http://orientdb.com/.

[9] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1103–1114, 2014.

[10] Alibaba Cloud. Double 11 real-time monitoring system with time series database. https://www.alibabacloud.com/blog/594855, 2019.

[11] Renzo Angles. The property graph database model. In *AMW*, 2018.

[12] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[13] Marco Arnaboldi, Jean-Pierre Lozi, Laurent Phillipe Daynes, Vlad Ioan Haprian, Shasank Kisan Chavan, Kapp Hugo, and Sungpack Hong. Parallel and efficient technique for building and maintaining a main memory, CSR-based graph index in an RDBMS, August 17 2021. US Patent 11,093,459.

[14] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 583–598, 2016.

[15] Martin Boissier. Reducing the footprint of main memory HTAP systems: Removing, compressing, tiering, and ignoring data. In *Proceedings of the VLDB 2018 PhD Workshop*, 2018.

[16] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[17] Shaosheng Cao, XinXing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. TitAnt: Online real-time transaction fraud detection in Ant Financial. *Proceedings of the VLDB Endowment*, 12(12):2082—-2093, August 2019.

[18] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446, 2004.

[19] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. ByteHTAP: ByteDance's HTAP system with high data freshness and strong data consistency. *Proc. VLDB Endow.*, 15(12):3411–3424, 2022.

[20] Peter P. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[21] Rong Chen and Haibo Chen. Wukong: A distributed framework for fast and concurrent graph querying. *ACM SIGOPS Operating Systems Review*, 55(1):77–83, 2021.

[22] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, Eurosys '15, pages 1–15, 2015.

[23] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, Eurosys '16, page 26, 2016.

[24] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, Eurosys '12, page 85–98, 2012.

[25] Dean De Leo and Peter Boncz. Packed memory arrays - rewired. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 830–841, 2019.

[26] Dean De Leo and Peter Boncz. Teseo and the analysis of structural dynamic graphs. *Proceedings of the VLDB Endowment*, 14(6):1053–1066, 2021.

[27] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. TigerGraph: A native MPP graph database. *arXiv preprint arXiv:1901.08248*, 2019.

[28] Christian Fahrner and Gottfried Vossen. A survey of database design transformations based on the entity-relationship model. *Data Knowl. Eng.*, 15(3):213–250, 1995.

[29] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. GraphScope: A unified engine for big graph processing. *Proceedings of the VLDB Endowment*, 14(12):2879–2892, 2021.

[30] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. RisGraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the 2021 International Conference on Management of Data*, pages 513–527, 2021.

[31] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *Proceedings of the 7th International Conference on Learning Representations*, ICLR '19, 2019.

[32] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi. CSR++: A fast, scalable, update-friendly graph data structure. In *24th International Conference on Principles of Distributed Systems*, OPODIS '20, 2020.

[33] Per Fuchs, Domagoj Margan, and Jana Giceva. Sortledton: A universal, transactional graph data structure. *Proceedings of the VLDB Endowment*, 15(6):1173–1186, 2022.

[34] 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 *11th USENIX symposium on operating systems design and implementation*, OSDI '14, pages 599–613, 2014.

[35] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[36] Henry Haselgrove. Wikipedia page-to-page link database. http://haselgrove.id.au/wikipedia.htm, 2010.

[37] Mohamed S Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G Aref, and Mohammad Sadoghi. Extending in-memory relational database engines with native graph support. In *Proceedings of the 21st International Conference on Extending Database Technology*, EDBT '18, pages 25–36, 2018.

[38] 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, August 2020.

[39] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. HyPer: Adapting columnar main-memory data management for transactional and query processing. *IEEE Data Eng. Bull.*, 35(1):46–51, 2012.

[40] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, ICLR '17, 2017.

[41] Pradeep Kumar and H Howie Huang. GraphOne: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)*, 15(4):1–40, 2020.

[42] 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*, pages 591–600, 2010.

[43] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 31–46, 2012.

[44] Per-Ake Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-Time analytical processing with SQL Server. *Proc. VLDB Endow.*, 8(12):1740–1751, August 2015.

[45] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proc. VLDB Endow.*, 10(12):1598–1609, August 2017.

[46] Chaojie Li, Wensen Jiang, Yin Yang, Shirui Pan, Gang Huang, and Lijie Guo. Predicting best-selling new products in a major promotion campaign through graph convolutional networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[47] Feifei Li. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.

[48] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. G-SQL: Fast query processing via graph exploration. *Proceedings of the VLDB Endowment*, 9(12):900–911, 2016.

[49] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374, 2015.

[50] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 37–50, 2017.

[51] Daniel Martin, Oliver Koeth, Johannes Kern, and Iliyana Ivanova. Near real-time analytics with IBM DB2 analytics accelerator. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 579–588, 2013.

[52] MySQL. MySQL internals manual: Chapter 20 the binary log. https://dev.mysql.com/doc/internals/en/binary-log.html.

[53] Sen Pan, Menghan Xu, Pei Yang, Lipeng Zhu, Aihua Zhou, and Jing Jiang. Research on application scenarios of HTAP in distribution network. In *2021 IEEE Sustainable Power and Energy Conference (iSPEC)*, pages 3916–3920, 2021.

[54] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1372–1385, 2021.

[55] Marcus Paradies, Cornelia Kinder, Jan Bross, Thomas Fischer, Romans Kasperovics, and Hinnerk Gildhoff. GraphScript: Implementing complex graph algorithms in SAP HANA. In *Proceedings of The 16th International Symposium on Database Programming Languages*, pages 1–4, 2017.

[56] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-Time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, 2018.

[57] Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. SnappyData: A hybrid transactional analytical store built on Spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2153–2156, 2016.

[58] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. Adaptive HTAP through elastic resource scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2043–2054, 2020.

[59] Noa Roy-Hubara, Lior Rokach, Bracha Shapira, and Peretz Shoval. Modeling graph database schema. *IT Prof.*, 19(6):34–43, 2017.

[60] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-Store: A real-time OLTP and OLAP system. In *Proceedings of the 21th International Conference on Extending Database Technology*, EBDT '18, 2018.

[61] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, dec 2017.

[62] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607, 2018.

[63] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 433–448, 2019.

[64] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. Accelerating analytical processing in MVCC using fine-granular high-frequency virtual snapshotting. In *Proceedings of the 2018 International Conference on Management of Data*, pages 245–258, 2018.

[65] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. Retrofitting high availability mechanism to tame hybrid transaction/analytical processing. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, pages 219–238, 2021.

[66] Sijie Shen, Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. DrTM+B: Replication-driven live reconfiguration for fast and general distributed transaction processing. *IEEE Transactions on Parallel and Distributed Systems*, 33(10):2628–2643, 2022.

[67] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 317–332, 2016.

[68] John Stegeman. Native vs. non-native graph database. https://neo4j.com/blog/native-vs-non-native-graph-technology/.

[69] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. SQLGraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[70] Qiaoyu Tan, Ninghao Liu, Xing Zhao, Hongxia Yang, Jingren Zhou, and Xia Hu. Learning to hash with graph neural networks for recommender systems. In *Proceedings of The Web Conference 2020*, pages 1988–1998, 2020.

[71] The Transaction Processing Council. TPC-C benchmark v5.11. http://www.tpc.org/tpcc/.

[72] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md Shahidul Haque Apu, and Huijuan Peng. IBM DB2 Graph: Supporting synergistic and retrofittable graph queries inside IBM DB2. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 345–359, 2020.

[73] 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, 2021.

[74] 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*, pages 839–848, 2018.

[75] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. FlexGraph: a flexible and efficient distributed framework for GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 67–82, 2021.

[76] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[77] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. GraphWalker: An I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, pages 559–571, 2020.

[78] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *Proceedings of 18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, 2021.

[79] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 233–251, 2018.

[80] Brian Wheatman and Randal Burns. Streaming sparse graphs using efficient dynamic sets. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 284–294, 2021.

[81] Xiating Xie, Xingda Wei, Rong Chen, and Haibo Chen. Pragh: Locality-preserving graph traversal with split live migration. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 723–738, 2019.

[82] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeff Naughton, and John Cieslewicz. F1 Lightning: HTAP as a service. *Proc. VLDB Endow.*, 13(12):3313–3325, August 2020.

[83] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. GNNLab: A factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 417–434, 2022.

[84] Kangfei Zhao and Jeffrey Xu Yu. All-in-One: Graph processing in RDBMSs revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1165–1180, 2017.

[85] Ai-Hua Zhou, Li-Peng Zhu, Meng-Han Xu, Sen Pan, Jun-Feng Qiao, Hong-Bin Qiu, and Song Deng. Research on mixed transaction analytical data management oriented to data middle platform. In *2021 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 308–312, 2021.

[86] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 301–316, 2016.

[87] 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, 2020.

Comosum: An Extensible, Reconfigurable, and Fault-Tolerant IoT Platform for Digital Agriculture

Gloire Rubambiza
Cornell University
gloire@cs.cornell.edu

Shiang-Wan Chin
Cornell University
sc2983@cornell.edu

Mueed Rehman
Cornell University
mr2265@cornell.edu

Sachille Atapattu
Cornell University
sa2257@cornell.edu

José F. Martínez
Cornell University
martinez@cornell.edu

Hakim Weatherspoon
Cornell University
hweather@cs.cornell.edu

Abstract

This work is an experience with a deployed networked system for digital agriculture (or DA). DA is the use of data-driven techniques toward a sustainable increase in farm productivity and efficiency. DA systems are expected to be overlaid on existing rural infrastructures, which are known to be less robust than urban infrastructures. While existing DA approaches partially address several infrastructure issues, challenges related to data aggregation, data analytics, and fault tolerance remain open. In this work, we present the design of Comosum, an extensible, reconfigurable, and fault-tolerant architecture of *hardware*, *software*, and *distributed cloud* abstractions to sense, analyze, and actuate on different farm types. We also present FarmBIOS, an implementation of the Comosum architecture. We analyze FarmBIOS by leveraging various applications, deployment experiences, and network differences between urban and rural farms. This includes, for instance, an edge analytics application achieving 86% accuracy in vineyard disease detection. An eighteen-month deployment of FarmBIOS highlights Comosum's tolerance to intermittent network outages that lasted for several days during many periods of the deployment. We offer practical insights on how FarmBIOS adapts to new DA vendors, reconfigurability challenges in the cloud, persistent failures that are unique to the DA context, and the system's current limitations.

1 Introduction

Digital agriculture (DA) is the use of data-driven techniques towards a "sustainable intensification" [65] in farm productivity and efficiency. DA is the next generation stemming from *precision* agriculture, which is local, offline, precise application of farm inputs (e.g., water, fertilizer, etc.) [44, 57, 58]. In contrast, DA systems involve more complex data processing and communication, both on and off rural farms [79]. DA systems are expected to be overlaid on existing rural infrastructures. However, rural infrastructures (e.g., Internet, power) are known to be less robust than urban infrastructure [17]. This is due, in part, to sparse populations [41], urban-centered technology design and standards [17], frequent outages [42], and limited maintenance [27]. These challenges make networked data aggregation and analytics on rural farms difficult [88].

While state-of-the-art approaches address several of these DA issues [26, 38, 85, 88], a lot of challenges related to data aggregation, data analytics, and fault tolerance remain open. First, although the diversity of sensor providers is growing [4, 5, 80, 83, 84, 87], data aggregation is difficult because of distributed data sources, incompatible sensors and data formats, and software dependencies. Second, the set of data analysis methods are increasingly leveraging advanced techniques such as machine-learning (ML) [45, 53, 68]; however, existing data analysis platforms rarely account for the variety of sensing mechanisms and crop types (e.g., row crop vs specialty farms). Third, state-of-the-art platforms [78, 88] partially address rural Internet and power challenges. However, fault tolerance is difficult to achieve across heterogeneous devices, networks, and cloud services. Overcoming these challenges requires extensibility, reconfigurability, and fault tolerance in the (1) underlying sensors and networking, (2) overlaying software, and (3) supporting cloud services.

In this paper, we present Comosum,[1] a cloud-based hardware/software architecture that takes a significant step toward this goal. Comosum is designed for researchers integrating heterogeneous DA platforms. To address the heterogeneity, Comosum relies on prior strong systems concepts such as separation of devices and device drivers [73], modularity [28, 43, 61], and reconfigurability of closed-source software/hardware [18]. Our general approach to DA-enabled farms is modular with abstractions for *hardware* (i.e., sensors and networking devices), *software*, and the *distributed cloud* (i.e., a cloud that combines the edge cloud at the farm, the public cloud, and sensor vendor clouds).

Specifically, the Comosum design presents three principles:

- **Extensible modules**: Comosum modules (§§ 3.2) pro-

[1] Named after Chlorophytum comosum, also known as spider plants, for their extensible leaves and adaptability to many conditions [67].

vide an abstraction on the acts of sensing, storing, computing, and actuating farm data. This abstraction is derived from the object-oriented programming idea of inheritable *instances*, which can be customized for different DA applications. Note that these modules are oblivious to the underlying hardware. This design consideration follows from the principle of dumb switches and smart control planes in Software-Defined Networking (SDN) [18]. In this manner, given a uniform API, the hardware (i.e., sensing and networking devices) and software modules can evolve independently.

- **Fault-tolerant distributed cloud**: The Comosum distributed cloud (§§ 3.3) addresses challenges in supporting elastic, vendor-neutral, and fault-tolerant data aggregation/analytics. Specifically, we highlight the complex data path where some data must be pulled directly from the sensor vendor, despite farm networking challenges. This need for networked data aggregation across the farm (or "edge") cloud, public (or "core") clouds, and private (or "vendor") clouds distinguishes the consequential fault tolerance trade-offs that are unique to DA environments (e.g., plants might die if not irrigated on time). Ultimately, by reimagining prior approaches such as CloneCloud [19], the Comosum design enables offline data collection and edge analytics during network outages.

- **Reconfigurable control plane**: Given software abstractions and distributed cloud deployment environments, the Comosum control plane (§§ 3.4) coordinates inter-module communication. To maintain reconfigurability and extensibility to heterogeneous devices, this component draws from the separation of devices and device drivers, inter-process communication (IPC), and SDN. Specifically, Comosum modules rely on a message-passing [89, 92] protocol to abstract away the distributed deployment environments.

We implemented a version of Comosum that we call FarmBIOS. We deployed multiple FarmBIOS instances in the Azure, AWS, and Google clouds. Further, we deployed these instances on one commercial farm and two research farms. Commercial farms are for-profit; in contrast, research farms are associated with land-grant universities [1, 9]. As a result, we operate multiple deployments (both in open fields and greenhouses) throughout the year with different Comosum module, cloud, and control plane configurations. The configurations differ such that they can meet the needs of animal (§§ 5.1), row crop (§§ 5.3), and specialty farms (§§ 5.2).

The results based on deployments and evaluation show that Comosum supports extensible, reconfigurable, and fault-tolerant DA systems. First, we applied FarmBIOS instances to a plant water stress application, as well as two ML-based applications that yielded 86% and 97% training accuracy in vineyard and dairy cow disease detection, respectively (§ 5). Second, we present an 18-month deployment with a million sensor readings from 80 sensors networked over cutting-edge hardware in two farm edge clouds (§§ 5.3). We further analyze Comosum's reconfigurability trade-offs based on our extensive deployment experience, application requirements, and comparative spectrum measurements of a 55-acre urban farm and a 615-acre rural farm (Appendix B). Third, we show Comosum's adaptability to faulty sensors in the field (§§ 6.2), its tolerance to a network outage over multiple days (§§ 6.1) at the edge, and the edge analytics' resilience to network outages (§§ 5.2). Deploying Comosum yielded several surprising architectural insights. First, failures at the edge (i.e., in the sensors and networking) vary differently from failure towards the core of the cloud (i.e., Internet and cloud module failures), and the differences in failure scope had to be identified, escalated, and optionally tolerated or repaired. Intermittent network failures were tolerated through offline data collection at the edge, or by directly performing analytics at the edge. Long-term sensor failures were escalated by digital twins. A digital twin is a digital representative of a physical object that behaves like its real-world counterpart. That is, we expanded the failure scope to cloud-based digital twins of sensors and notified human operators. We call this an *active* digital twin. A key takeaway was that the time to detect and tolerate a system component failure varied from seconds to weeks. Second, frequent unannounced API or data format changes by vendors led to many errors. Comosum evolved to shield DA application developers from this complexity by providing a uniform API; specifically, a unstructured platform layer and structured application layer. Third, Comosum was made cloud provider-agnostic by providing a cloud-independent layer (i.e., through cloud-agnostic abstractions such as tables/functions) and a separate cloud-dependent layer (through cloud-dependent services such as Azure Table and AWS Simple Notification Service).

We present an experience with a deployed networked system for DA, our contributions are as follows:

- Three case studies to demonstrate DA research challenges, and how they motivate our design goals
- An integrative approach that applies and advances the state-of-the-art to a portfolio of system challenges associated with building Comosum effectively across multiple subsystems and contexts (Table 3)
- The design and implementation of Comosum, a cloud-based architecture that unifies state-of-the-art DA approaches under a single interface by distilling largely complex black-box technologies down to classical ideas
- Evidence that Comosum supports various applications, tolerates intermittent network failures, and can be deployed across different farm types and cloud providers

Paper outline: § 2 describes three DA case studies to motivate design goals. § 3 delves into the Comosum system

software architecture. § 4 instantiates FarmBIOS, a Comosum implementation that unifies otherwise incompatible DA systems. § 5 describes three FarmBIOS applications and deployment contexts. § 6 describes system adaptations to ensure long-term maintenance. § 8 puts Comosum in the context of prior work before concluding in § 9. Appendix B demonstrates Comosum's reconfigurability trade-offs.

2 Challenges: Why DA is Hard

Comosum is borne out of four years of collaborative research building state-of-the-art systems to support DA. We describe the main challenges in building on existing technologies to motivate the architecture developed in § 3.

2.1 Challenge 1: Data Aggregation

Data aggregation involves pulling and processing data from a variety of sources. Consider, for example, DA researchers in animal science who need real-time monitoring of dairy cows to facilitate early disease detection [33, 34, 54, 72]. This requires integrating data from wearable and non-wearable cow sensors, herd management software, and manual data collection on site.

Existing wearable sensors capture behavioral, physiological, and performance parameters such as physical activity, rumination and eating time, estrous behavior, and internal temperature (e.g., [2, 3]). Non-wearable sensors include cameras, weather stations, milk meters and near-infrared spectrometers, and weight scales that deliver milk yield and body weight from a static location as cows pass through with every milking session (e.g., [4, 80, 84]). The data monitoring is possible via systems from various sensor providers running on the farm computer (e.g., [5, 83, 87]). The providers control and avail the data for download either as raw files (via FTP dumps) or JSON (through scripted API calls to the providers' servers).

However, data aggregation efforts face five major hurdles. First, the sensor data are siloed in monolithic platforms with *incompatible APIs*. Second, the datasets are delivered in various *incompatible formats* (e.g., Excel, JSON, or DIF). Third, the data is *distributed* between the farm computer and numerous sensor providers' servers. Fourth, the patchy data integration programs are *dependent* on the farm computer's operating system (e.g., to run PowerShell scripts) and disk storage layout (e.g., hardcoded, per-provider directories). Fifth, any ad hoc integration protocol is likely to introduce *data sparsity* as new sensors are incorporated and old sensors are retired. These hurdles highlight the need to aggregate sensor data from heterogeneous devices and platforms. *The system should enable data aggregation from arbitrary sensor vendor platforms, data formats, and data locations*.

2.2 Challenge 2: Data Analytics

Data analytics involves extracting actionable insights from big data. It is important in managing and predicting farm inputs and expected outputs, respectively. Consider, for example, DA researchers in plant pathology who need fast methods to detect vineyard diseases affecting grape and wine quality [7, 22, 68, 93]. This requires detecting symptoms typically visible on the leaves.

Existing grapevine disease detection methods include molecular tests, remote sensing, and digital models. Molecular tests involve plucking diseased leaves to be analyzed in laboratories. Remote sensing and digital models combine aerial imagery from Unmanned Aerial Vehicles (UAV), vegetation indices, and machine learning (ML) techniques [45, 52, 53, 68].

However, these existing data analytics approaches face three challenges. First, current remote sensor data cleaning and pooling processes, which are often *manual*, do not enable cross-farm analytics. Secondly, ML models have been shown to produce *contradictory analyses* depending on the choice of hyperparameters [15, 20, 81]. Lastly, molecular tests are *slow* (i.e., on the order of days) to yield actionable results in a setting where every second implies further disease spread. These challenges highlight the need for a reconfigurable approach for data storage and model training to enable fast iterations in any environment. *The system should enable fast plug-and-play of different analytics modules and sensing mechanisms*.

2.3 Challenge 3: Fault Detection/Tolerance

Fault tolerance involves detecting, recovering from, and optionally repairing system faults. It is important in providing timely manual or automated interventions when farm monitoring assets such as sensors have failed. Consider, for example, DA researchers in plant breeding who need to understand water status effects on plant growth by controlling for variables such as temperature, soil moisture, and CO2 levels [49, 71, 86]. This requires reliable sensor data collection with both short-term and long-term storage, processing, and actuation.

Existing systems (e.g., [78, 88]) generally comprise sensing hubs, an Internet gateway device, and optional cloud storage and processing. The sensor hubs communicate to the gateway over various protocols such as unlicensed TV White Spaces (TVWS) [88] or ZigBee [40]. The gateway relays the sensor data over various media (e.g., 4G/3G [40], WiFi [85]) to diverse cloud-based routing hubs (e.g., Azure IoT Hub [88], AWS IoT Core [46]) for long-term storage. With a few exceptions [26, 46], prototype deployments are often outdoors [85, 88].

However, fault detection and tolerance are difficult due, in part, to three challenges related to cascading failures. First, faulty sensors affect data collection. Second, network outages affect data storage and data processing both locally at the farm and in the cloud. Third, the complexity and heterogeneity of existing hardware/software systems pose significant troubleshooting issues in the field. Individually, these failures can present real consequences in farms (e.g., plants are not irrigated, animals are not fed, etc.). This highlights the importance of fault tolerance and detection in DA environments. *The system should detect, localize, tolerate and/or repair failures in sensor, network, and software components*.

2.4 Summary and Design Goals

These case studies highlight interoperability challenges within state-of-the-art systems. These challenges map to three core system design goals:

- **Extensibility**: In addressing data aggregation (§§ 2.1), we aim to provide an *extensible interface* that can generalize sensing, analytics, and actuation across APIs, clouds, hardware, and platforms.
- **Reconfigurability**: In addressing data analytics (§§ 2.2), we aim to *allow different points in the configuration space* towards data models that can be trained and used across different networking and cloud deployment scenarios.
- **Fault Tolerance**: In addressing fault detection/tolerance (§§ 2.3), we aim to *detect and/or tolerate intermittent failures* despite the heterogeneity of hardware, farm types, and cloud services.

3 Comosum Architecture

3.1 Overview

In this section, we describe Comosum's *hardware*, *software* and *distributed cloud* - the main components in the quest to sense, analyze, and actuate on rural farms (see Figure 1).

Hardware encompasses sensing, networking, and control devices. Sensing devices produce updates based on changes in real-world conditions such as temperature, soil moisture, vegetation density, etc. Networking devices provide infrastructure support via data routing and transfer between other devices, of which routers, switches, and antennas are typical representatives. Control devices offer digital interfaces with programmable logic control (PLC) functions. While programmable, control devices are limited to firmware running on a few kilobytes of memory.

Software encompasses possible manipulations of data generated by the hardware entities. The software modules, which we describe from left to right as shown in Figure 1, offer abstractions on the acts of sensing, storing, computing, and intervening based on real-world changes. The changes are communicated via interrupt and poll mechanisms (see Table 1). The telemetry module serves as an interface to capture and reflect physical and virtual state changes. Whereas physical updates read directly from sensing devices (e.g., GPIO pins), virtual updates are third-party reports (e.g., Excel files) from vendors on proprietary sensors (see §§ 3.3). The storage module is a logical abstraction over storage structures (databases, files, etc.) and formats (Excel, CSV, etc.). The compute module captures the various networked, temporal, and spatial arrangements of compute devices (see *Cloud* below) to produce actionable results. Together, the storage and compute modules form the analytics module. Henceforth, the analytics module is used interchangeably with the storage and compute modules. Lastly, the actuation module bridges the analytics module to control devices and farm operators.

Distributed cloud encompasses ubiquitous, convenient, on-demand network access [60] to storage media and compute devices at the farm edge, the public cloud, and the private cloud (i.e., sensor vendor servers, university servers, etc.). Storage media vary from low-end USB sticks to a pool of hard disk drives in the cloud. Compute devices span low-end Raspberry Pis at the edge to high-end virtual machines (VMs).

To achieve reconfigurability and extensibility, Comosum draws inspiration from Software Defined Networks (SDNs) [18]. Here, the data plane spans the hardware and software. The control plane then is the custom configuration process of hardware components and software modules to solve individually unique DA problems (e.g., Case 1-3 in § 2). Borrowing from the object-oriented design paradigm (OOP), the fundamental Comosum unit is the *instance*.

Comosum instance modules operate in an event-driven approach with per-module processes [89]. Independent, message-passing [92] modules have two advantages. First, this enables modules to be deployed anywhere in the distributed cloud. Secondly, it simplifies reasoning about application correctness for multi-threaded modules. For instance, irrigation should be triggered in the actuation module only after a dry forecast is observed in the compute module.

To summarize, each Comosum instance is a configuration of sensor, compute, storage, and actuation modules to map and solve different sets of real-world agricultural challenges. In the following subsections, we describe in depth the extensible Comosum modules, the Comosum distributed cloud architecture and its vendor neutrality goal (§§ 3.3), and the Comosum control plane's reconfiguration capabilities (§§ 3.4).

3.2 Comosum Modules

Telemetry Module. The telemetry module is the entry point into a Comosum instance's data plane. The telemetry module requires that any interested parties (observers) are notified of new sensor readings. To that end, we design the module as an abstract class following the observer design pattern [61]. The interface comprises *register*, *notify*, *read*, and *run* operations. In managing observers, the observer argument is an abstract data type (ADT). Upon being notified, observers receive a message indicating the state change. The message is similarly an ADT, and it is used to receive new updates through an invocation of the telemetry module's *read* method. By using an ADT, the module enables chained updates where downstream modules may serve as observables (other telemetry modules) and observers (any abstraction implementing the observer pattern). In practice, sensor updates are consumed by the analytics module.

Storage Module. The storage module is inspired by the classic UNIX [73] file system interface with simple *read* and *write* operations. To align with another prevalent storage model, the change feed, the module additionally supports

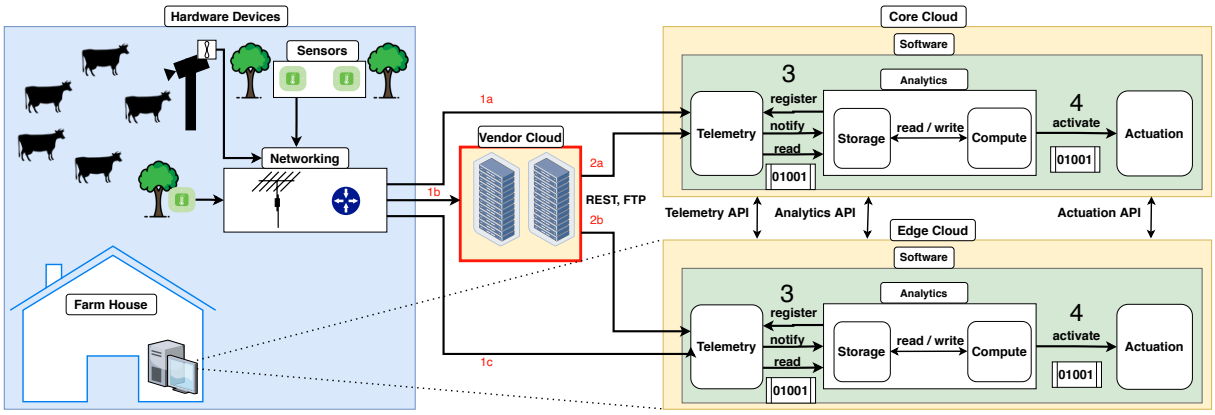


Figure 1: The Comosum distributed cloud partitions modules between the farm ("edge"), remote ("core"), and sensor vendor clouds.

change_feed and *next* methods that operate on iterators of new data (inserts, updates). Besides mimicking UNIX and database semantics, we strive for transparent access of stored objects [76]. In other words, similar to the observer argument in the telemetry module, the path argument in reads and writes is an ADT. This makes the storage module extensible to storage calls of various structures and data formats. Finally, to accommodate modules operating with topic-based storage services, the storage module supports the publish/subscribe paradigm. The pub-sub interface enables other modules to push data in addition to subscribing to updates based on topics of interest. Thus, the storage module exposes *subscribe*, *push*, *pull*, and *notify* operations.

Compute Module. The compute module operates on state changes from the telemetry and storage modules. Therefore, the module similarly follows the observer design paradigm. Acting as an observer, the compute interface supports four operations, namely *notification_sensor_rcv*, *notification_storage_rcv*, *analytics*, and *run*. The first two can be invoked to receive data based on previous compute module registrations and subscriptions to the telemetry and storage modules, respectively. The *analytics* method receives new procedure calls off the wire, and it uses the metadata and data to execute local/remote application logic. Optionally, the compute module invokes the storage module's *write* or *push* methods to store intermediary results.

Actuation Module. The actuation module is the final endpoint in the data plane. Like the sensor module, the actuation module provides an interface to physical operation of and virtual notification to real-world entities. That is, the actuation is either automated or mediated. For automated actuation, devices between the farmhouse and the field issue command messages. In this context, the module issues *activate* calls to the appropriate control devices. The command argument optionally identifies the device to execute the command. Upon an *activate* invocation in a mediated actuation scenario, the actuation module effectively is a wrapper for push notification services such as text messages and email.

3.3 The Comosum Distributed Cloud

The motivating case studies (§ 2) demonstrated that the sources of data consumed by DA applications greatly varies. While the sensor data is mostly generated locally, the initial storage and compute operations are executed on-demand by remote servers which are often owned by the sensor providers. Thus, another challenge for Comosum is achieving *vendor neutrality*. That is, hardware/software innovations must not be tied to a particular cloud-based service vendor.

To that end, we introduce the Comosum distributed cloud abstraction to support elastic, vendor-neutral sensing, storage, compute, and actuation capabilities. The Comosum distributed cloud (Figure 1) consists of the farm (or "edge"), public (or "core"), and private (or "vendor") clouds. The vendor cloud maps the more complex data path where some data must be pulled from sensor vendor servers instead of directly from the sensor abstracted away by Comosum. Unlike the "edge" and "core" clouds, Comosum modules cannot be deployed in the vendor clouds. Further, the Comosum distributed cloud design is similar in motivation to CloneCloud [19]. CloneCloud, which offloads computation from remote devices to the cloud, assumes an always-available, more powerful compute pool in the cloud. In contrast, Comosum enables computation and fault tolerance at the edge (remote device) when the (core) cloud is unavailable or too expensive to use.

While the *edge cloud*, *core cloud* and *vendor cloud* separation meets the resource elasticity and vendor neutrality goals, it also collides with limited bandwidth at farms.

On one hand, the edge cloud provides storage and computation closer to the data source. For applications with ephemeral storage and computation needs, this eliminates core cloud connectivity and latency challenges. Specifically, by leveraging networking advances such as LoRa [8] and TVWS [12], the edge cloud is capable of completely disconnected operation in deployments with large variations in area and granularity. Conversely, the edge cloud is generally unsuitable for storage- and compute-intensive Comosum applications (e.g. §§ 2.2).

On the other hand, the core and vendor clouds offer sig-

Module	Interface Method	In/Out?	Int/Poll?	Description
telemetry	register(observer)	Input	Int	Add sensor update observer
	notify(update)	Output	Int	Notify observers of a new update
	read(update)	Output	Poll	Read latest update
	run()	Input	Int	Run the sensor module
storage	write(path, data)	Input	Int	Write to storage medium
	read(path)	Output	Poll	Read from a storage medium
	change_feed()	Input	Poll	Offer an iterator to new data
	next(iterator)	Output	Poll	Get the next record from an iterator
	subscribe(subscriber, topics)	Input	Int	Register a new subscriber
	push(topic, data)	Input	Int	Publish new updates to storage
	notify(subscriber, data)	Output	Int	Push new updates to subscribers
	pull(topic)	Output	Poll	Pull recent updates, if any
	run()	Input	Int	Run the storage module
compute	notification_sensor_rcv(context)	Input	Int	Receive data from sensor update
	notification_storage_rcv(new_data)	Input	Int	Receive data from storage subscription
	analytics(arg)	Input	Int	Execute application business logic
	run()	Input	Int	Run the compute module
actuation	activate(cmd)	Output	Int	Execute a command on a control device
	run()	Input	Int	Run the actuation module

Table 1: The unified Comosum API is inspired by classic system design approaches (e.g., design patterns [61], UNIX file system [73]).

nificantly more storage and computation capacity, albeit at a higher network latency cost. Therefore, faced with 'reliably unreliable' [42] Internet at remote locales, an application whose progress relies on a consistent connection to the core and vendor clouds is bound to fail. Losing connection to cloud-based time critical decisions risks real consequences for farmers; plants may perish from water stress; cows may die from preventable diseases or difficult births; and vast vineyard swaths may succumb to virus infection.

In summary, the *edge cloud*, *core cloud*, and *vendor cloud* separation meets Comosum's resource elasticity and vendor neutrality goals. Note, however, that it also reveals difficult system trade-offs; for instance, latency in the context grapevine disease detection (§§ 5.3).

3.4 The Comosum Control Plane

An important Comosum goal is that initial design allows for the integration of devices and software modules. This is the guiding principle of the Comosum control plane. The control plane draws inspiration from device drivers, IPC, and SDNs. Comosum leverages a diverse array of networking and storage primitives. These primitives in turn define extensible libraries and configuration templates that accommodate communication between devices and software modules from current and future DA systems.

On the hardware front, sensing devices require wrappers for new serial device drivers or wrappers to existing standard interfaces (e.g., RS485 [69]). Networking devices necessitate new packet processing interfaces. Finally, control devices typically expose wrappers for their PLCs.

On the software and cloud components, the control plane specifies inter-module communication. Due to the distributed nature of Comosum modules, we use message passing [89,92] where modules communicate as follows.

Modules call a dispatcher with a message specifying the peer module to contact. The dispatcher in turn maps message queues to socket connections to the peer modules. In this scheme, the modules remain oblivious to the underlying networking. Further, this simplifies part of the control plane to IP address tuples which can be edited as the underlying network changes. While communication between Comosum modules is sockets-based, any module calls to other systems (e.g sensor vendor clouds) uses whatever higher-level abstraction that the external systems expose (e.g., REST APIs).

As discussed in the overview (§§ 3.1), a key Comosum goal is to easily reconfigure devices and software modules to fit different applications. Therefore, Comosum implementations must remain as close to a set of reusable configuration and compilation templates as possible. Configuration templates include cloud connection strings, sensor SKUs, IP addresses, etc. Compilation templates include Docker files [24], remote procedure call (RPC) definition files, package dependencies, etc. Thus, the hardware and software components are reconfigurable to solve various DA challenges.

4 FarmBIOS: A Comosum Implementation

The particular instantiation of the Comosum architecture presented here is the Farm Basic Input Output System (FarmBIOS), drawing inspiration from its unification of routinely incompatible hardware and software systems. Henceforth, we use FarmBIOS and Comosum interchangeably. FarmBIOS code and research datasets are open source (Appendix A).

We built FarmBIOS in Python, atop Google's protocol buffers (also known as protobufs) [35]. Protobufs provide a language-independent, efficient serialization protocol that allows not only the construction of Comosum modules in numerous languages, but also extensibility of RPC templates to enable integration with arbitrary DA systems. Figure 2 shows the FarmBIOS stack. Next, we briefly describe the most salient components of FarmBIOS instances.

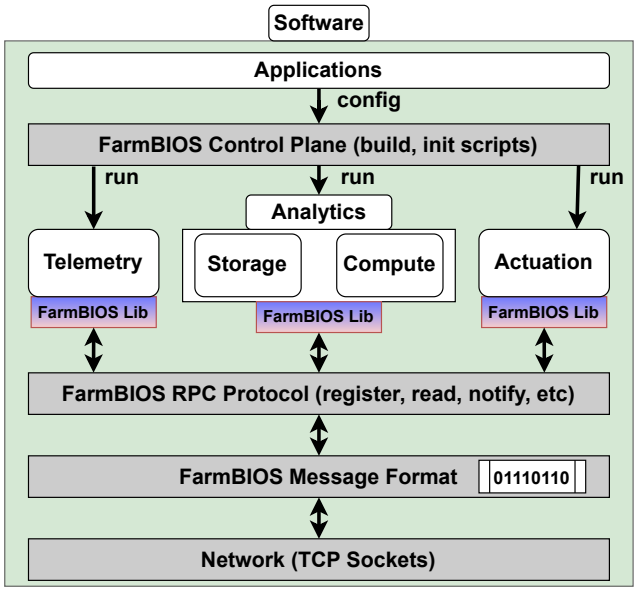


Figure 2: FarmBIOS - an implementation of Comosum

4.1 FarmBIOS Control Plane

At the top of the FarmBIOS stack, applications specify software configurations to the FarmBIOS control plane, which (1) provisions the appropriates edge and core cloud compute and storage resources (e.g., VMs, databases), (2) instantiates the required modules, and (3) builds any required Docker containers. Based on the application specific requirements, the modules are deployed and started in the edge or core cloud. Note, however, that not all modules in the middle layer are required by every application. These à la carte application configurations are at the heart of Comosum's vendor neutrality (e.g., pairing Azure compute and AWS storage).

4.2 FarmBIOS Library

The FarmBIOS library (FarmBIOS Lib) is the implementation of the Comosum module abstraction. That is, the library allows the customization of base telemetry, storage, compute, and actuation module classes to fit the purposes and configurations of different applications. Practically the library deals with challenges related to perpetual deprecation of hardware in the field and software libraries. The FarmBIOS Lib addresses this data processing challenge by providing wrappers to an array of services such as tables, databases, CSV readers, and email clients. In the current implementation, we provide wrappers around the Azure Table [62], Azure CosmosDB SQL [63], Azure Machine Learning Workspace (Azure ML) [64], Twilio [48], and OpenWeather [47] services in addition openpyxl [32] and CSV readers - with more templates to be added as our array of supported applications expands. Therefore, FarmBIOS Lib is a *structured, cloud-independent* application layer.

4.3 FarmBIOS RPC Protocol

FarmBIOS is built in a *network-agnostic* manner. This implementation choice is important for extensibility to arbitrary DA platforms. Specifically, the modules are unaware of differences between local and remote peer modules. By local we mean modules operating within the same host. Local and remote operations entail modules passing and receiving messages to/from their dispatcher. The dispatcher is tasked with routing the procedure call to the appropriate peer module based on the control plane configuration. The RPCs rely on a client-server architecture built on TCP sockets wrapped by a selector operating per-connection queues, and each connection tunnels to a peer module. In both scenarios, any data communication occurs over the common FarmBIOS RPC protocol. Note that the underlying (TCP) communication protocol is known only to the dispatcher, but not the modules. Further, the RPC protocol makes no assumptions on the data formats, thereby maintaining data introspection/formatting flexibility for applications through FarmBIOS Lib.

4.4 FarmBIOS Message Format

The Comosum modules exploit OOP, UNIX, IPC, and other intuitive semantics. In practice, however, the underlying implementations address two major obstacles. First, the number of data formats, which represent technical compatibility negotiations between research farms and sensor providers, are both unwieldy and subject to unexpected changes [23]. Therefore, similar to the Linux file abstraction, Comosum offers a uniform, byte-addressable format for inter-module data communication - the FarmBIOS Message Format. This format is an *unstructured* platform layer. Secondly, host operating systems eventually get upgraded or lose long term support. Comosum achieves independence from the host OS by leveraging application orchestration tools such as Docker [24] and Kubernetes [29].

5 Applications & Deployment Experiences

In this section, we describe the hardware contexts, FarmBIOS usage (§ 4), and deployment contexts of the motivating challenges (§ 2) to showcase Comosum's range of applications. Specifically, the applications are different Comosum configurations that meet the needs of animal farms (§§ 5.1), specialty farms (§§ 5.2), and row crop farms (§§ 5.3).

5.1 CowsOnFitbits

Building on Challenge 1 (§§ 2.1), CowsOnFitbits is a data aggregation component supporting early disease prediction models which achieve 97% training accuracy [59]. The edge

Training	Data Location	Data Size	Compute	Storage	Runtime	Accuracy
Edge	Local	4MB	8 CPUs	256GB	27.1s	84%
Edge	Cloud	-	8 CPUs	256GB	35.6s	86%
Azure ML	Cloud	-	2 vCPUs	100GB	86.5s	86%

Table 2: Grapevine disease detection: model training runtime and accuracy for various WineGuard configurations

cloud is a farm PC (16GB RAM, 500GB storage) running the Windows 10 Enterprise OS. The OS features a Docker engine deployed on its Windows Subsystem for Linux (WSL). The edge is connected to vendor clouds and a university cloud via a 1Gbps Ethernet connection.

CowsOnFitbits leverages FarmBIOS modules (packaged as Linux containers) as follows. Sensor reports are made available by the providers via FTP dumps to the edge cloud. The telemetry module continuously polls the local disk, awaiting the FTP dumps. New reports trigger the telemetry module's *notify* function call which, in turn, notifies its local compute module. The telemetry module's *read* method is called by the compute module. The compute module aggregates sensor data from multiple streams to be stored in the private university cloud, where a module exposes a REST API for data access and queries by ML applications. In the cloud, cows are identifiable across data streams through farm-unique cow ID's. The storage calls are made to an intermediate, non-FarmBIOS module operating a Cassandra database [70].

CowsOnFitbits has been actively tracking approximately 1,500 cows in a commercial farm for nearly three years; having collected 23GB of datasets at the time of writing. The testing/deployment experience with CowsOnFitbits offers two observations. First, the unstructured platform layer (§§ 4.4) is more stable than the structured application layer (§§ 4.2). Specifically, unannounced API/format changes on the vendor side, which routinely occur every few weeks, introduce breaking changes in FarmBIOS Lib. The API breaking changes affected the API to the *vendor-specific* application layer changes (i.e., breaks). FarmBIOS tolerates these API changes by insulating itself with *vendor-independent* layers that use unstructured files to store data from the vendor along with methods that can interpret the unstructured data according to the latest vendor interface definition. The CowsOnFitbits system adapted to a recent change in approximately one day. This required minor application changes and deploying a new Docker container. Subsequent versions may benefit from Kubernetes [29], especially its rollbacks and canary deployments. Second, we observed missing/duplicate data in the core cloud due to mismatched interpretation of floating points in protobufs vs Python, missing vendor reports, unexpected signal interference between RFID tags and electrical engines for manure systems, or farm workers tripping over wires. The floating point issue was resolved through meticulous end-to-end testing of FarmBIOS modules over the course of a year. The missing report and power outage issues remain as open issues, though we propose a potential fix (see § 6, § 7).

5.2 WineGuard

Building on Challenge 2 (§§ 2.2), WineGuard is a data analytics platform for grapevine disease detection; achieving up to 86% training accuracy. WineGuard's sensor data originates from plane flights over California vineyards in September 2020 using NASA's Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) [36]. The spectroscopic sensor data is publicly accessible from the NASA cloud [66]. The AVIRIS data is merged with disease ground truth data from molecular tests on select leaves from the same period. The merged data are uploaded to the Azure Cloud for experimental retrieval.

WineGuard employs FarmBIOS modules as follows. We built a wrapper for Azure ML [64] to kickstart a reconfigurable model training pipeline. Note that the application can similarly be built using equivalent tools from other cloud providers such as Amazon SageMaker [77]. The model training configuration includes Azure subscription ID strings, the name of a pre-provisioned workspace, the location of the training data, and a text file with required Python packages. Upon issuing the compute module's *analytics* command, the configuration is deployed for training.

We deployed the WineGuard training pipeline in an edge cloud and in the Azure Cloud (see Table 2) [2]. This deployment presents several insights. First, the model accuracy is relatively stable regardless of training location, and, as expected, training at the edge with local data incurs the least runtime. Second, there is a 31% runtime overhead when fixing the location at the edge. We observed that this is due to an initial "warm-up" of the training data download from the cloud. In the best case, cloud-based training and inference are instantaneous. Otherwise, during disconnected periods, inference can be done faster and locally at the edge. Third, the satellite coordinate data were occasionally off by a few meters compared to the ground truth disease data. Without requiring manual intervention, analytics on sparse data in near real-time was necessary to correct the errors. In particular, FarmBIOS enabled the rectification of the divergence by selecting only spectroscopic bands with reliable data while still enabling the training/inference to proceed.

5.3 WaterGuard

Building on Challenge 3 (§§ 2.3), WaterGuard is a plant water stress alert system for research farms. WaterGuard is prototyped with research software and hardware provided by

[2]We tried to use a Raspberry Pi 4B (4 CPUs, 32GB storage) as the edge cloud. However, the ARM processor could not execute Docker containers built on an x86-64 architecture (the alternative edge used here) [14]. Building on an armv7 base image also failed due to end of support for distro updates.

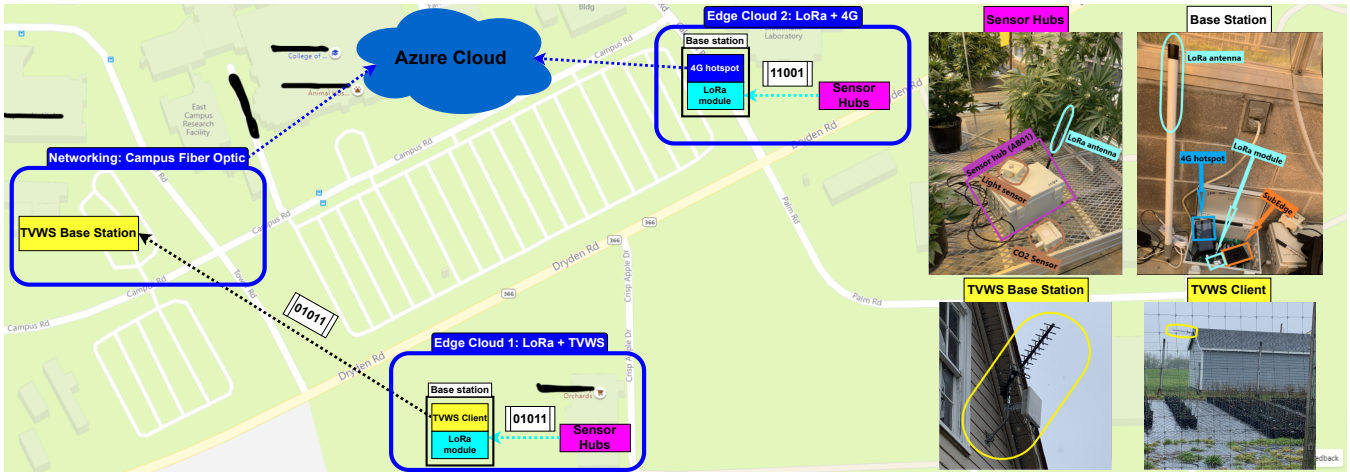


Figure 3: The Comosum hardware repertoire deployed for 18 months. Edge Cloud 1 (active since October 2020) serves an apple orchard water stress study. Edge Cloud 2 (active since March 2021) serves high-throughput corn, hemp, and strawberry breeding experiments.

Microsoft Research [88] (Figure 3). The hardware features sensor hubs from Seeed Studio operating on eight D-size batteries and supporting up to 13 analog and digital sensors. The sensor readings are networked over LoRa [8] to a base station comprised of an edge device, a LoRa module, and LoRa antenna (5dBi, 900MHz). The edge device is a general purpose UpBoard (4GB RAM, 32GB storage) running the Windows 10 IoT Core OS. In addition to the LoRa components, the device is connected to a TVWS Client (6Harmonics Inc), which provides physical and data link connectivity over a single TV channel (18, 497MHz) to a TVWS Base Station (TVBS) located at a research barn approximately a quarter of a mile away. Finally, the TVBS is wired to the university's 1Gbps fiber-optic Internet as a gateway for sensor data to Azure.

WaterGuard relies on FarmBIOS modules as follows. Sensor readings are relayed to Azure. Note that Azure table storage offers no *change feed* for observer notifications. Thus, the telemetry module relies on periodic *reads* (i.e., polling the storage module) to detect inserts that, in turn, should trigger its *notify* method. The analytics unit is notified via a *notification_sensor_rcv* call to the compute module. Next, based on the configuration received from the new update, the compute module *reads* from the shared storage module to get data on the appropriate sensor hub and start the *analytics*. Finally, upon reaching an irrigation decision, the compute module calls *activate* on the actuation module which notifies the researchers over text message using the Twilio API [48].

WaterGuard has been deployed for 18 months in two edge clouds with nine sensor hubs (Figure 3). Each sensor hub averages eight sensors and 223 days of data collection. Together, the hubs have collected over a million sensor readings. The biggest insight from this deployment was the unexpected mundane work required to adapt experimental DA systems ([88]) to new settings where failures can happen anywhere in the sensor-to-cloud continuum (see § 6).

6 Adapting to the Wild

The previous section described the hardware configurations, API usage, and deployment experiences/insights from three FarmBIOS instances. Here, we present the successes and system adaptations necessary for long-term Comosum maintenance. Though the adaptations are specific to WaterGuard, the key idea of *active digital twins* is broadly applicable.

6.1 Offline Data Collection Is Not Enough

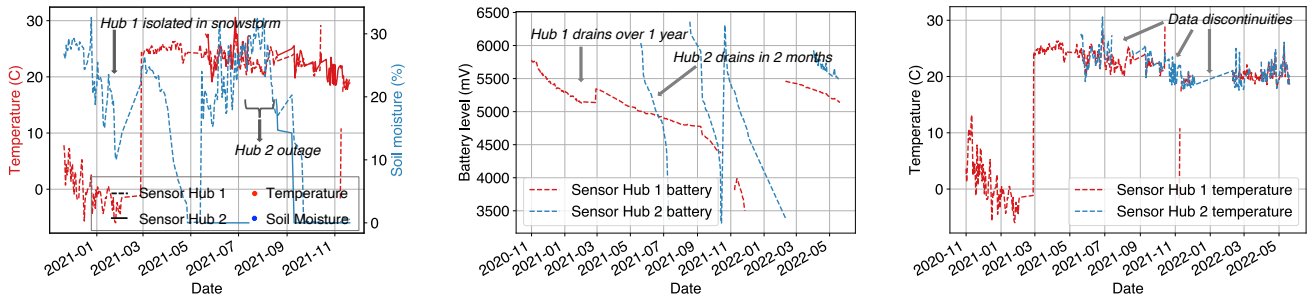
WaterGuard is capable of tolerating days-long network outages by relying on offline data collection and standard hardware redundancy. As shown in 4a, the pilot sensor hub (Sensor Hub 1) achieved disconnected operation during a snow storm and heat wave in February 2021 and May 2021, respectively. Until the 4G hotspot connectivity was restored, the sensor data was simply stored at the edge device.

However, data aggregation and analytics are still affected by faulty sensors and human error. Concretely, faulty CO2 sensors drained the batteries faster than expected (4b) and/or slight misconfigurations place data in incorrect columns. Both faults effectively result in data discontinuities (4c).

6.2 The Fix: Active Digital Twins

To streamline fault detection, Comosum evolved to include reactive monitoring [90]. That is, detecting, escalating, and optionally repairing system faults at different failure scopes. The Comosum design easily lends itself to this task by introducing ***active* digital twins.** A digital twin is a digital representation of a physical object, process, or environment that behaves like its real-world counterpart. In contrast, *active* digital twins combine the traditional digital twin concept with Comosum's actuation design.

The active digital twins were an emergent concept as we iterated over FarmBIOS to make it more fault-tolerant, especially in outdoor research farm deployments where a delayed



(a) Network outages (b) Faulty sensors (c) Data discontinuities

Figure 4: Adapting FarmBIOS to (a) network outages, (b) faulty sensors, and (c) data discontinuities

detection could imply troubleshooting sensor hubs in two feet of hardened snow (Figure 4a, [75]). When the digital twin diverges from its physical twin beyond a reconfigurable margin (e.g., five minutes), an action is taken by the system such as sending notifications to human operators. In Comosum, only the sensor hubs are twinned. We modeled timely data collection, and divergence from the physical system is characterized by missing telemetry over a 30 minute period.

Concretely, we implemented a web-based, reconfigurable notification system with three key functions. The Status page leverages the WaterGuard sensor hubs' digital twins to display their connection state and last activity. The Configuration page provides an interface for retrieving and editing each sensor hub's port configurations. Lastly, the Notifications page provides a reconfigurable set of functions including (1) whether the notification system is **enabled**, (2) an editable **threshold** (in minutes) for triggering outage notifications, (3) an editable **frequency** for outage checks, and (4) the set of **emails** to be notified during outages. Note that the notification system relies on data aggregation and analytics across multiple, separate cloud services (i.e., Azure IoT Hub, Azure Table, AWS Simple Notification Service), and the implementation simply plugs Comosum module implementations of these services with minimal or no change to other modules. Therefore, it is broadly applicable to detect failures at different stages from the sensors at the farm to the modules in the cloud.

7 Practical Insights and Limitations

The motivating challenges (§ 2) highlighted major interoperability issues within state-of-the-art DA platforms (see Table 3 for a summary). We mapped these challenges to three core system design goals: *extensibility*, *reconfigurability*, and *fault tolerance*. We summarize below both persistent and new lessons from our experiences, and, more importantly, how FarmBIOS practically addressed the challenges. We also state the system's current limitations.

- **Extensibility to new DA vendors comes with (minor) costs**: Table 3 details the challenges in DA data aggregation. Our design goal was to provide an *extensible interface* that can generalize across APIs, clouds, hardware, and platforms. FarmBIOS provides a unified interface to merge/analyze/actuate datasets spread across the distributed cloud. For instance, CowsOnFitbits (§§ 5.1) enables the merging of datasets from four different vendors. The tradeoff is that, for each new vendor, a new script (less than 50 lines of code) must be written to move vendor reports to the appropriate directories for FarmBIOS module triggers.

- **The cloud surprisingly complicates reconfigurability**: Table 3 details difficulties in system reconfigurations to support different farm networking and analytics pipelines. Our design goal was to *allow different points in the configuration space* towards data models that can be trained and used across different networking and cloud deployment scenarios. Indeed, we explored numerous hardware/software configuration possibilities (see Appendix B). However, as observed in WineGuard (§§ 5.2), the interface between the cloud and the long-lived deployed systems was not stable. In particular, the Azure ML APIs were subject to parameter deprecations which affect the WineGuard compute configurations. FarmBIOS evolved around these externalities by treating incoming parameters in telemetry and analytics modules as abstract data types. The result is a stable platform that shields users from these volatilities.

- **Failure in DA systems is the norm, not the exception**: Table 3 details the heterogeneity and failure cases that complicate DA system deployments and maintenance. Our design goal was to *detect and/or tolerate intermittent failures* despite the heterogeneity of hardware, farm types, and cloud services. In CowsOnFitbits (§§ 5.1) for instance, we observed missing data due to sensing/networking failures (frequency interference between sensors and manure systems), human factors (tripping over wires), etc. In another instance (WineGuard, (§§ 5.2)), analytics on sparse data is a necessity for deployed DA systems. We demonstrate how FarmBIOS copes with the unreliability of the underlying sys-

Why It Is Hard	System Challenge	Design/Implement Decisions	Supporting Results & Contributions
Data Aggregation	Incompatible platforms	Reuse classic ideas (§§ 3.1, §§ 3.2)	Unified platform API (§ 5)
	Incompatible formats	Byte-addressable payloads (§§ 4.4)	Extensible libraries (§§ 4.2)
	Distributed data	Mask data path/sources (§§ 3.3)	Merged across clouds (§§ 5.1)
	Host dependence	Containerization (§§ 4.4)	Cross-architecture transfer (§§ 5.1)
		RPC dispatcher (§§ 4.3)	Cross-platform transfer (§§ 5.1)
Data Analytics	Manual processing	Comosum design (§ 3)	Automated processing (§§ 5.2)
	Unreproducible models	Reconfig. models (§§ 3.3, §§ 3.4)	Distributed training (§§ 5.2)
	Data sparsity	Active digital twins (§§ 6.2)	Divergence detection (§§ 5.2)
	Slow actuation	Comosum design (§ 3)	Sub-minute inference (§§ 5.2)
Fault tolerance	Faulty sensors	Standard hardware redundancy (§§ 6.1)	18-month deployment (§§ 5.3)
		Active digital twins (§§ 6.2)	Reconfig. notification system (§§ 6.2)
	Network outages	Reconfig. control plane (§§ 3.4)	Reconfig. networks (§§ 5.2, Appendix B)
		Edge analytics design (§§ 3.3)	Edge inference (§§ 5.2)
		Offline data collection (§§ 3.3)	Tolerate 7-day outage (§§ 6.1)
	Complexity/heterogeneity	Modularity (§§ 3.2)	Comosum design (§ 3)

Table 3: A summary mapping of systems challenges to Comosum design decisions and their supporting results

tems through the broadly applicable idea of *active digital twins*. In the case of frequency interference, for example, the RFID sensors could be twinned. In the vineyard context, the error message indicating mismatch between ground truth and satellite data could be used as input for a digital twin model.

- **FarmBIOS evolution and limitations**: As emphasized above, failure is the rule and not the exception. FarmBIOS was designed to tolerate failure, and as stated in §§ 2.3, plant and livestock depend on a robust system. Therefore, the system was improved over time. For instance, the active digital twin implementation relied on the telemetry module abstraction to increase fault tolerance. One limitation is the *lack of support for automated movement of computations between the edge and core clouds during permanent power/network outages*. Recall, however, that the edge side of the architecture is, in fact, capable of operating autonomously, at a minimum to retrieve and store sensor data (§§ 5.3) and, if so configured, perform local computation (§§ 5.2). We leave this limitation for future endeavors.

8 Related work

8.1 Programming Frameworks

To streamline application development and partitioning for resource-constrained environments, recent community efforts leverage the cloud for mobile and IoT applications. By rewriting application executables, the CloneCloud [19] architecture intelligently partitions program portions for dynamic execution between mobile devices and their cloud twins. The partitioner identifies expensive application portions through static and dynamic code analysis that informs an optimizer to solve the execution partitioning challenge. Along with EdgeProg [55] and like CloneCloud, the TinyLink [26,39] systems form a set of cloud-native, generative systems of hardware configurations and software executables for IoT applications. TinyLink and EdgeProg expose high-level APIs and If-This-Then-That (IFTTT) languages to abstract away the low-level knowledge for developers, respectively.

In line with the prior work, Comosum exposes high-level APIs for interfacing with IoT platforms without deep knowledge of the underlying networking and hardware. Unlike CloneCloud, Comosum partitions applications at the module level, not the instruction level. Departing from EdgeProg's use of IFTTT and TinyLink's exclusive support of application development in C-like languages, Comosum supports module development with any language compatible with the (de)-serialization protocol shared by the modules.

8.2 Agricultural Sensor Networks

The rise of low-cost IoT sensor networks has led to an explosion of new communication standards and protocols being ported to industrial and consumer applications. For example, like WaterGuard, Gutiérrez *et al.* [40], Ahmad *et al.* [6], and Vasisht *et al.* [88] showcase the application of GPRS, XBEE, and TVWS technologies to agricultural monitoring systems, respectively. Further, Ayoub *et al.* [11] and Jawad *et al.* [50] present detailed overviews of both power-hungry (e.g., WiFi, Bluetooth, etc.) and low power wide area network (LPWAN) technologies (e.g., LoRa, NB-IoT, etc.) and their recent applications to, among others, dairy health care, automation, and greenhouse monitoring. Comosum demystifies these novel networking technologies' potential and limitations to interdisciplinary audiences interested in similar applications.

Besides IoT networking standards, the literature identifies open challenges in IoT networking, hardware, and software (co)design. The most salient include extensibility [82], durability [13], reliability [37,82], modularity [82], scalability [37], energy efficiency [39,82,88], and interoperability

among heterogeneous devices [82]. The Comosum design addresses extensibility, durability, reliability, modularity, configurability, and interoperability. Further, scalability is indirectly addressed through decoupling and thin APIs that allows independent evolution of the software modules and the underlying networking hardware, protocols, and devices.

Perhaps closest to our agricultural application of edge computing is Taneja *et al.*'s SmartHerd management system [85]. Like SmartHerd, Comosum co-opts a microservice approach, wherein the sensing, compute, storage, and actuation modules can seamlessly be placed in bandwidth rich as well as constrained environments. Further in line with SmartHerd, Comosum easily aggregates data from incompatible sensor vendors' private web servers to avoid vendor lock-in. Still, § 5 shows FarmBIOS's extensibility beyond dairy applications.

8.3 IoT Architecture Abstractions

Sisinni *et al.* [82] define a *reference IoT architecture* as a "higher level of abstraction description that helps identify issues and challenges for different application scenarios". This definition reflects the three years of exploration that resulted in the Comosum architecture. Previous architectural approaches identify sensing or perception [10, 25, 51, 91], physical [56], interface [91], networking [10, 25, 51, 91], transport [56], middleware [56], and service or application [10, 25, 51, 56, 91] layers as essential to an IoT application.

Although the architectures fundamentally serve applications with different business and technical needs, their essential layers are modules in the Comosum design. Thus, the benefit of Comosum is its partition of the physical *hardware* from the *software* that manipulates the networked data. That is, the *software*, by acting as a collection of byte-passing modules, is extensible because it is agnostic to the evolution, intricacies, protocols, or any other factors of the hardware.

9 Conclusion

In this paper, we present Comosum, a system software architecture to support digital agriculture (DA) applications in research and commercial farms. The architecture comprises *hardware*, *software*, and *distributed cloud* abstractions to build *extensible*, *reconfigurable*, and *fault-tolerant* sensor networks for farms. By supporting diverse DA applications in multiple clouds, we show that FarmBIOS, a Comosum implementation, meets these design goals. Eighteen months of Comosum instance deployments and adaptations reveal new insights on fault-tolerant sensor networks for DA. We introduce *active* digital twins to streamline fault detection, escalation, and optional repair from sensors to cloud-based modules. In ensuring that systems approaches employed in urban research farms readily map to rural farm realities, a thorough analysis highlights practical insights, limitations, and trade-offs (see Appendix) that are unique to DA applications as a starting point for community discussions of DA's potential contributions to networked system design and implementation beyond the current state-of-the-art.

Acknowledgments

This work was funded in part by a Microsoft Cornell Digital Agriculture Summer Research Fellowship, a Microsoft Investigator Fellowship, the US National Science Foundation (NSF) (Grants #1922551, #2019674, and #1955125), the U.S. Department of Agriculture (USDA) (Grants #2017-67015-26772 and #2023-77038-38865). The authors appreciate deployment maintenance and software development efforts from the Center for Advanced Computing (CAC) at Cornell University (especially Chris Myers and Brandon Barker), Kevin Huang, Yifan Zhao, and Annie Kimmel. Comosum benefited from expertise and logistical support from our collaborators in Animal Science (Julio O. Giordano and Martin M. Pérez), Plant Pathology (Fernando Emiliano Romero Galván, and Kaitlin M. Gold), and Plant Science (Savanah Dale, Mike Gore, Nicholas Kaczmar, Neil Mattson) to realize the CowsOnFibits, WineGuard, and WaterGuard applications, respectively. The networking infrastructure for this project would not have been possible without the help from numerous Information Technology professionals (especially Scott Yoest, Dora Abdullah, and Avery Quinn Smith), building managers (especially Scott Albrecht), orchard managers, and carpenters at Cornell University. The authors would like to thank Ken Birman, Kevin Negy, A. F. Cooper, Danny Adams, Cody Sunderlin, the anonymous reviewers, and our shepherd, Sándor Laki, for their constructive feedback and insightful discussions during the manuscript's preparation and artifact evaluation. Lastly, the authors appreciate the support from Ranveer Chandra, Elizabeth Bruce, and Tusher Chakraborty at Microsoft for providing access to cloud credits and guidance in deploying the FarmBeats platform.

References

[1] 1890FOUNDATION, . Our History - Land Grant and Universities: A Primer. https://www.1890foundation.org/history-of-land-grant-universities.

[2] AFIMILK. AfiAct II: The Leading Cow Leg Sensor. https://www.afimilk.com/cow-monitoring#afi-collar, Nov. 2021.

[3] AFIMILK. AfiCollar: Advanced Neck Collar for Cow Monitoring. https://www.afimilk.com/cow-monitoring#afi-collar, Nov. 2021.

[4] AFIMILK. AFILAB. https://www.afimilk.com/afilab/, Nov. 2021.

[5] AFIMILK. mySilent Herdsman. https://my.silentherdsman.com/\#/login, Nov. 2021.

[6] AHMAD, N., HUSSAIN, A., ULLAH, I., AND ZAIDI, B. H. IOT based Wireless Sensor Network for Precision Agriculture. In 2019 7th International Electrical Engineering Congress (iEECON) (2019), pp. 1–4.

[7] AL-SADDIK, H., SIMON, J.-C., AND COINTAULT, F. Assessment of the optimal spectral bands for designing a sensor for vineyard disease detection: The case of 'Flavescence dorée'. Precision Agriculture 20, 2 (2019), 398–422.

[8] ALLIANCE, L. A Technical View of LoRa and LoRaWAN. LoRa Alliance (2015).

[9] APLU, A. Land-Grant University FAQ. https://www.aplu.org/about-us/history-of-aplu/what-is-a-land-grant-university/.

[10] ATZORI, L., IERA, A., AND MORABITO, G. The Internet of Things: A survey. Computer Networks 54, 15 (2010), 2787–2805.

[11] AYOUB, W., SAMHAT, A. E., NOUVEL, F., MROUE, M., AND PRÉVOTET, J.-C. Internet of Mobile Things: Overview of LoRaWAN, DASH7, and NB-IoT in LPWANs Standards and Supported Mobility. IEEE Communications Surveys Tutorials 21, 2 (2019), 1561–1581.

[12] BAHL, P., CHANDRA, R., MOSCIBRODA, T., MURTY, R., AND WELSH, M. White Space Networking with Wi-Fi like Connectivity. In Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (New York, NY, USA, Aug. 2009), SIGCOMM '09, Association for Computing Machinery, pp. 27–38.

[13] BAUER, J., AND ASCHENBRUCK, N. Design and Implementation of an Agricultural Monitoring System for Smart Farming. In 2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany) (2018), pp. 1–6.

[14] BIRADAR, P. Standard_init_linux.go:211: Exec user process caused "exec format error". https://stackoverflow.com/questions/58298774/standard-init-linux-go211-exec-user-process-caused-exec-format-error.

[15] BOUTHILLIER, X., LAURENT, C., AND VINCENT, P. Unreproducible Research is Reproducible. In Proceedings of the 36th International Conference on Machine Learning (June 2019), K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97 of Proceedings of Machine Learning Research, PMLR, pp. 725–734.

[16] BROADBANDNOW. Expert Overview of HughesNet's Services. https://broadbandnow.com/HughesNet, May 2021.

[17] BURRELL, J. Thinking Relationally about Digital Inequality in Rural Regions of the US. First Monday 23, 6 (2018).

[18] CASADO, M., KOPONEN, T., SHENKER, S., AND TOOTOONCHIAN, A. Fabric: A Retrospective on Evolving SDN. In Proceedings of the First Workshop on Hot Topics in Software Defined Networks (2012), pp. 85–90.

[19] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. CloneCloud: Elastic Execution between Mobile Device and Cloud. In Proceedings of the Sixth Conference on Computer Systems (New York, NY, USA, 2011), EuroSys '11, Association for Computing Machinery, pp. 301–314.

[20] COOPER, A. F., LU, Y., FORDE, J. Z., AND SA, C. D. Hyperparameter Optimization Is Deceiving Us, and How to Stop It. In Advances in Neural Information Processing Systems (2021).

[21] DB, R. T. W. Channel Search. https://usa.wavedb.com/channelsearch/tvws, May 2021.

[22] DI GENNARO, S. F., BATTISTON, E., DI MARCO, S., FACINI, O., MATESE, A., NOCENTINI, M., PALLIOTTI, A., AND MUGNAI, L. Unmanned Aerial Vehicle (UAV)-based remote sensing to monitor grapevine leaf stripe disease within a vineyard affected by esca complex. Phytopathologia Mediterranea 55, 2 (2016), 262–275.

[23] DILORENZO, J., ZHANG, R., MENZIES, E., FISHER, K., AND FOSTER, N. Incremental Forest: A DSL for Efficiently Managing Filestores. SIGPLAN Not. 51, 10 (Oct. 2016), 252–271.

[24] DOCKER. Docker Personal: Get Started with Docker for Free. https://www.docker.com/products/personal, Nov. 2021.

[25] DOMINGO, M. C. An overview of the Internet of Things for people with disabilities. Journal of Network and Computer Applications 35, 2 (2012), 584–596.

[26] DONG, W., LI, B., GUAN, G., CHENG, Z., ZHANG, J., AND GAO, Y. TinyLink: A Holistic System for Rapid Development of IoT Applications. ACM Trans. Sen. Netw. 17, 1 (Sept. 2020).

[27] DUARTE, M. E., VIGIL-HAYES, M., ZEGURA, E., BELDING, E., MASARA, I., AND NEVAREZ, J. C. As a Squash Plant Grows: Social Textures of Sparse

Internet Connectivity in Rural and Tribal Communities. ACM Transactions on Computer-Human Interaction 28, 3 (July 2021), 16:1–16:16.

[28] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. Exokernel: An operating system architecture for application-level resource management. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (New York, NY, USA, Dec. 1995), SOSP '95, Association for Computing Machinery, pp. 251–266.

[29] FOUNDATION, C. N. C. Production-Grade Container Orchestration. https://kubernetes.io/, Nov. 2021.

[30] GALLARDO, R., AND WHITACRE, B. A Look at Broadband Access, Providers and Technology. Tech. rep., Purdue Center for Regional Development, West Lafayette, IN, USA, 2019.

[31] GARNETT, P., AND ROBERTS, S. Overview of Internet service provider technology considerations for rural broadband deployments. Tech. rep., Microsoft Airband Initiative Team, Redmond, WA, USA, 2018.

[32] GAZONI, E., AND CLARK, C. Openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files, Jan. 2020.

[33] GIORDANO, J., PEREZ, M., RIAL, C., NYDAM, D., YOU, Y., WANG, Y., AND WEINBERGER, K. Improving dairy cow health monitoring and management using automated sensors. In Conference of Research Workers in Animal Diseases. Abs (2021), vol. 447, p. 346.

[34] GOKUL, V., AND TADEPALLI, S. Implementation of smart infrastructure and non-invasive wearable for real time tracking and early identification of diseases in cattle farming using IoT. In 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC) (2017), pp. 469–476.

[35] GOOGLE. Protocol Buffers, Feb. 2010.

[36] GOWEY, K. Airborne Visible / Infraraded Imaging Spectromer (AVIRIS NG) DATA. https://avirisng.jpl.nasa.gov/data.html, Nov. 2020.

[37] GRGIĆ, K., ŽAGAR, D., BALEN, J., AND VLAOVIĆ, J. Internet of Things in Smart Agriculture — Possibilities and Challenges. In 2020 International Conference on Smart Systems and Technologies (SST) (2020), pp. 239–244.

[38] GUAN, G., FU, K., CHENG, Z., GAO, Y., AND DONG, W. Rapid development of IoT applications with TinyLink. In 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS) (May 2017), pp. 956–957.

[39] GUAN, G., LI, B., GAO, Y., ZHANG, Y., BU, J., AND DONG, W. TinyLink 2.0: Integrating Device, Cloud, and Client Development for IoT Applications. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking (New York, NY, USA, 2020), MobiCom '20, Association for Computing Machinery.

[40] GUTIÉRREZ, J., VILLA-MEDINA, J. F., NIETO-GARIBAY, A., AND PORTA-GÁNDARA, M. Á. Automated Irrigation System Using a Wireless Sensor Network and GPRS Module. IEEE Transactions on Instrumentation and Measurement 63, 1 (Jan. 2014), 166–176.

[41] HARDY, J., WYCHE, S., AND VEINOT, T. Rural HCI Research: Definitions, Distinctions, Methods, and Opportunities. Proc. ACM Hum.-Comput. Interact. 3, CSCW (Nov. 2019).

[42] HASAN, S., BARELA, M. C., JOHNSON, M., BREWER, E., AND HEIMERL, K. Scaling Community Cellular Networks with CommunityCellularManager. In 16th ${$USENIX$}$ Symposium on Networked Systems Design and Implementation (NSDI 19) (2019), pp. 735–750.

[43] HEISER, G., UHLIG, V., AND LEVASSEUR, J. Are virtual-machine monitors microkernels done right? ACM SIGOPS Operating Systems Review 40, 1 (Jan. 2006), 95–99.

[44] HIGGINS, V., BRYANT, M., HOWELL, A., AND BATTERSBY, J. Ordering Adoption: Materiality, Knowledge and Farmer Engagement with Precision Agriculture Technologies. Journal of Rural Studies 55 (Oct. 2017), 193–202.

[45] HRUŠKA, J., ADÃO, T., PÁDUA, L., MARQUES, P., PERES, E., SOUSA, A., MORAIS, R., AND SOUSA, J. J. Deep Learning-Based Methodological Approach for Vineyard Early Disease Detection Using Hyperspectral Data. In IGARSS 2018 - 2018 IEEE International Geoscience and Remote Sensing Symposium (2018), pp. 9063–9066.

[46] IMTIAZ JAYA, N., AND HOSSAIN, M. F. A Prototype Air Flow Control System for Home Automation Using MQTT Over Websocket in AWS IoT Core. In 2018 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC) (Oct. 2018), pp. 111–1116.

[47] INC, O. Weather API. https://openweathermap.org/api, May 2021.

[48] INC, T. Programmable SMS. https://www.twilio.com/docs/sms/quickstart/python, May 2021.

[49] JAIN, P., LIU, W., ZHU, S., MELKONIAN, J., PAULI, D., RIHA, S. J., GORE, M. A., AND STROOCK, A. D. A minimally disruptive method for measuring water potential in-planta using hydrogel nanoreporters. bioRxiv (2020).

[50] JAWAD, H. M., NORDIN, R., GHARGHAN, S. K., JAWAD, A. M., AND ISMAIL, M. Energy-Efficient Wireless Sensor Networks for Precision Agriculture: A Review. Sensors 17, 8 (2017).

[51] JIA, X., FENG, Q., FAN, T., AND LEI, Q. RFID technology and its applications in Internet of Things (IoT). In 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet) (Apr. 2012), pp. 1282–1285.

[52] JIMÉNEZ-BRENES, F. M., LÓPEZ-GRANADOS, F., TORRES-SÁNCHEZ, J., PEÑA, J. M., RAMÍREZ, P., CASTILLEJO-GONZÁLEZ, I. L., AND DE CASTRO, A. I. Automatic UAV-based detection of Cynodon dactylon for site-specific vineyard management. PloS one 14, 6 (2019), e0218132.

[53] KERKECH, M., HAFIANE, A., AND CANALS, R. Vine disease detection in UAV multispectral images using optimized image registration and deep learning segmentation approach. Computers and Electronics in Agriculture 174 (2020), 105446.

[54] KIM, H., MIN, Y., AND CHOI, B. Real-time temperature monitoring for the early detection of mastitis in dairy cattle: Methods and case researches. Computers and Electronics in Agriculture 162 (2019), 119–125.

[55] LI, B., AND DONG, W. EdgeProg: Edge-centric Programming for IoT Applications. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS) (Nov. 2020), pp. 212–222.

[56] LIU, C. H., YANG, B., AND LIU, T. Efficient naming, addressing and profile services in Internet-of-Things sensory environments. Ad Hoc Networks 18 (2014), 85–101.

[57] LOWENBERG-DEBOER, J. The Precision Agriculture Revolution. Foreign Aff. 94 (2015), 105.

[58] M, A., AND S, I. Effects of Precision Irrigation on Productivity and Water Use Efficiency of Alfafa under Different Irrigation Methods in Arid Climates. Journal of Applied Sciences Research 7, 3 (2011), 299–308.

[59] M. PEREZ, M., YU, Y., WANG, Y., WEINBERGER, K. Q., NYDAM, D., AND O. GIORDANO, J. Performance of the machine learning method XGBoost for prediction of clinical health disorders in lactating dairy cows. Journal of Dairy Science 103, 1 (June 2020), 127–127.

[60] MELL, P., AND GRANCE, T. The NIST Definition of Cloud Computing. Tech. rep., National Institute of Standards and Technology, Gaithersburg, MD, USA, 2011.

[61] MICROSOFT. Observer Design Pattern. https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern.

[62] MICROSOFT. Get started with Azure Table storage and the Azure Cosmos DB Table API using Python. https://docs.microsoft.com/en-us/azure/cosmos-db/table-storage-how-to-use-python, May 2021.

[63] MICROSOFT. Quickstart: Build a Python application using an Azure Cosmos DB SQL API account. https://docs.microsoft.com/en-us/azure/cosmos-db/create-sql-api-python, May 2021.

[64] MICROSOFT. What is an Azure Machine Learning workspace. https://docs.microsoft.com/en-us/azure/machine-learning/concept-workspace, May 2021.

[65] MUELLER, N. D., GERBER, J. S., JOHNSTON, M., RAY, D. K., RAMANKUTTY, N., AND FOLEY, J. A. Closing yield gaps through nutrient and water management. Nature 490, 7419 (2012), 254–257.

[66] NASA. Enabling Earth Science in the Cloud. https://earthdata.nasa.gov/esds/cloud, Jan. 2021.

[67] NCSTATE, E. Chlorophytum comosum (Anthericum Comosum, Chlorophytum, Ribbon Plant, Spider Ivy, Spiderplant, Spider Plant, Walking Anthericum) | North Carolina Extension Gardener Plant Toolbox. https://plants.ces.ncsu.edu/plants/chlorophytum-comosum/.

[68] NGUYEN, C., SAGAN, V., MAIMAITIYIMING, M., MAIMAITIJIANG, M., BHADRA, S., AND KWASNIEWSKI, M. T. Early Detection of Plant Viral Disease Using Hyperspectral Imaging and Deep Learning. Sensors 21, 3 (2021), 742.

[69] NOTES, E. RS485 - an introduction. https://www.electronics-notes.com/articles/connectivity/serial-data-communications/rs485-introduction-basics.php.

[70] ORGANIZATION, A. Apache Cassandra | Apache Cassandra Documentation. https://cassandra.apache.org/_/index.html.

[71] PAGAY, V., M, S., A, S. D., J, H. E., O, V., A, P., N, C. T., N, L. A., AND D, S. A. A microtensionmeter capable of measuring water potentials below -10 MPa. Lab Chip 14, 15 (2014), 2806–2817.

[72] PEREZ, M., CABRERA, E., AND GIORDANO, J. Effect of automating health monitoring on detection of health disorders and performance of lactating dairy cows. Journal of Dairy Science 102 (2019), 24.

[73] RITCHIE, D. M., AND THOMPSON, K. The UNIX Time-Sharing System. Commun. ACM 17, 7 (July 1974), 365–375.

[74] RIZZATO, F. Mobile Experience in Rural USA- An Operator Comparison. https://www.opensignal.com/2019/09/24/mobile-experience-in-rural-usa-an-operator-comparison, Sept. 2019.

[75] RUBAMBIZA, G., SENGERS, P., AND WEATHERSPOON, H. Seamless visions, seamful realities: Anticipating rural infrastructural fragility in early design of digital agriculture. In Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (2022), pp. 1–15.

[76] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and Implementation of the Sun Network Filesystem. In Proceedings of the Summer USENIX Conference (1985), pp. 119–130.

[77] SERVICES, A. W. Amazon SageMaker. https://aws.amazon.com/sagemaker/, Nov. 2021.

[78] SERVICES, A. W. AWS IoT Greengrass: Build intelligent IoT devices faster. https://aws.amazon.com/greengrass/, Nov. 2021.

[79] SHEPHERD, M., TURNER, J. A., SMALL, B., AND WHEELER, D. Priorities for science to overcome hurdles thwarting the full promise of the 'digital agriculture' revolution. Journal of the Science of Food and Agriculture 100, 14 (2020), 5083–5092.

[80] SIMSHINE. Simshine Simam Alloy AS: Smart deterrence in the act. https://www.simcam.ai/simcam-alloy-1s, Nov. 2021.

[81] SINHA, K., PINEAU, J., FORDE, J., KE, R. N., AND LAROCHELLE, H. NeurIPS 2019 Reproducibility Challenge. ReScience 6, 2 (2020).

[82] SISINNI, E., SAIFULLAH, A., HAN, S., JENNEHAG, U., AND GIDLUND, M. Industrial Internet of Things: Challenges, Opportunities, and Directions. IEEE Transactions on Industrial Informatics 14, 11 (Nov. 2018), 4724–4734.

[83] SMAXTEC. BE SUCCESSFUL! WITH HEALTHY DAIRY COWS. https://smaxtec.com/en/.

[84] SUPPLY, G. T. Onset HOBO S-TMB-M006 Temperature Smart Sensor with 19.7' cable. https://www.globaltestsupply.com/product/onset-hobo-s-tmb-m006-temperature-smart-sensor, Nov. 2021.

[85] TANEJA, M., JALODIA, N., BYABAZAIRE, J., DAVY, A., AND OLARIU, C. SmartHerd management: A microservices-based fog computing–assisted IoT platform towards data-driven smart dairy farming. Software: Practice and Experience 49, 7 (2019), 1055–1078.

[86] VAN LEEUWEN, C., TRÉGOAT, O., CHONÉ, X., BOIS, B., PERNET, D., AND GAUDILLÈREJEAN-PIERRE. Vine water status is a key factor in grape ripening and vintage quality for red Bordeaux wine. How can it be assessed for vineyard management purposes? OENO One 43, 3 (Sept. 2009), 121–134.

[87] VAS. DairyComp: The World's Most Powerful Dairy Herd Management Tool. https://vas.com/dairycomp/, Nov. 2021.

[88] VASISHT, D., KAPETANOVIC, Z., WON, J., JIN, X., CHANDRA, R., SINHA, S., KAPOOR, A., SUDARSHAN, M., AND STRATMAN, S. FarmBeats: An IoT Platform for Data-Driven Agriculture. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17) (Boston, MA, Mar. 2017), USENIX Association, pp. 515–529.

[89] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (New York, NY, USA, 2001), SOSP '01, Association for Computing Machinery, pp. 230–243.

[90] WOOD, M. D., AND MARZULLO, K. The design and implementation of meta. Reliable distributed computing with the ISIS toolkit (1993), 309–327.

[91] XU, L. D., HE, W., AND LI, S. Internet of Things in Industries: A Survey. IEEE Transactions on Industrial Informatics 10, 4 (Nov. 2014), 2233–2243.

[92] YOUNG, M., TEVANIAN, A., RASHID, R., GOLUB, D., AND EPPINGER, J. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. SIGOPS Oper. Syst. Rev. 21, 5 (Nov. 1987), 63–76.

[93] ZARCO-TEJADA, P., POBLETE, T., CAMINO, C., GONZALEZ-DUGO, V., CALDERON, R., HORNERO, A., HERNANDEZ-CLEMENTE, R., ROMÁN-ÉCIJA, M., VELASCO-AMO, M., LANDA, B., ET AL. Divergent abiotic spectral pathways unravel pathogen stress signals across species. Nature Communications 12, 1 (2021), 1–11.

A Artifact Appendix

Abstract

The artifact documents our deployment experience and open-source efforts to build a **Software-Defined Farm** (or **SDF**) [75], also known as the **Comosum** or **FarmBIOS** system in the present paper. Comosum is intended to provide an extensible, reconfigurable, and fault-tolerant platform for IoT data collection, processing, and actuation. On one hand, the paper covers the architecture (Comosum), the implementation (FarmBIOS), and our deployment experiences over 18 months. On the other hand, the artifact provides a standalone Docker image and a pointer to the open-source code that, together, can be used to demonstrate the instantiation of the telemetry, analytics, and actuation concepts. In particular, the artifact demonstrates the repeatability of these ideas through three applications: CowsOnFitbits, WineGuard, and WaterGuard.

Scope

In addition to inspecting the code and research datasets, the artifact can be used as a starting point for extending the FarmBIOS/Comosum platform with new cloud services and sensor vendors. The current release of the artifact is intended to validate the listed claims about the applications:

- The CowsOnFitbits application (§§ 5.1) can use the Comosum sensor module (aka telemetry module) and compute modules to aggregate data from **three** (anonymous) IoT vendors and **six** data sources.
- The WineGuard application (§§ 5.2) can use can use the Comosum compute module abstraction to train machine learning models and perform local inference with at least 75% accuracy in approximately 30 seconds.
- The WaterGuard application (§§ 5.3) demonstrates the potential of *active digital twins* in increasing the platform's fault-tolerance to failures in the path from the sensors to the cloud.

Contents

The artifact includes a Docker image named *comosum-atc-artifact-eval* and a zipped archive of the code base built from the usenix-atc23-artifact-eval branch and the b313d7e commit in the SDF GitHub repository.

Hosting

The artifact is hosted on the publicly-funded archival platform Zenodo under this unique DOI.

Requirements

- The Docker image was primarily built and tested on an X86-based system (Windows 10 Education OS, Version 22H2, OS Build 19045.2846 with WSL 2 installed to emulate a Linux-like environment, and Docker Desktop Version 20.10.10). Therefore, the image should be loadable on most Unix-like environments with Docker installed.
- In addition to the primary development environment listed above, we reproduced the results on a X86-based system (Ubuntu Linux OS 22.04.1, Docker Version 23.0.6) and an arm64-based Macbook Pro (macOS Ventura 13.3, Docker Version 23.0.5)
- Although we have successfully reproduced the results on a Mac with an Apple Silicon (M2) chip, we cannot guarantee reproducibility if the evaluation is conducted on macOS, especially the M chips which are known to have issues with Docker Desktop filesystem change notifications and port mapping/forward issues. The Comosum system extensively relies on change notifications and port forwarding.

B Understanding the Trade-offs

This section presents an exploration of network configurations for Comosum applications with two goals in mind. First, we showcase Comosum's potential reconfigurability from a 55-acre urban farm to a 615-acre rural farm. Secondly, we offer a way for interdisciplinary DA researchers to quickly establish their networking needs by assessing three factors: expected application payload frequencies (§§ B.1), network availability and throughput in urban versus rural locations (§§ B.2), and desired system latency (§§ B.3). The key observation is that the DA context has the potential for new lessons and challenges to well-established networking, storage, and application management assumptions. The Comosum experiences serve a crucial starting point for the community conversation.

Application	Payload	Frequency	LoRa (SF:12)	DSL	Satellite	4G LTE	TVWS (1x6)	TVWS (4x6)	Fiber-optic
WaterGuard	65B	6 min	0.44 sec	5.20e-4 sec	1.73e-4 sec	9.60e-5 sec	5.20e-5 sec	2.80e-6 sec	5.20e-7 sec
WineGuard	4MB	Daily	7.60 hr	32.00 sec	10.67 sec	5.93 sec	3.20 sec	0.17 sec	0.03 sec
CowsOnFitbits	17MB	Daily	1.39 days	2.30 min	45.87 sec	25.48 sec	13.76 sec	0.74 sec	0.14 sec

Table 4: Estimated cloud backup time for FarmBIOS applications under various network bottleneck scenarios.

B.1 Application Data Rates

Table 5 illustrates the significant range of data generation rates for the three Comosum applications; from a few bytes every six minutes to hundreds of MBs weekly. The WaterGuard total is based on a sensor hub deployment with seven sensors (see §§ 5.3). The WineGuard dataset is based on spectrometer values from a 500m*300m vineyard field. The CowsOnFitbits total is an estimation based on data from four sensor providers tracking approximately 1,500 cows on a commercial farm. The CowsOnFitbits sensor data generation varies from every 10 minutes to once daily.

Application	Payload	Frequency
WaterGuard	65B	6 minutes
WineGuard	4MB	Daily
CowsOnFitbits	17MB	Varies

Table 5: FarmBIOS application data rates and formats.

B.2 Achievable Network Throughputs

We consider five networking media for data transfers at a farm, namely LoRa, DSL, Satellite, 4G LTE, and TVWS. Table 6 shows the achievable data transfer rates for the different media. The LoRa settings reflect current settings from the WaterGuard sensor hubs.
The DSL throughput is included because DSL holds the largest footprint in rural housing units' Internet access [30]. The TVWS settings reflect observed throughputs in the literature [31], measured TV channel occupations (as of Sept. 2020) at a campus research farm and a more rural farm 25 miles away, and tower height-based channel availability estimations [21] for the two farms. Based on their GPS coordinates, the campus farm offers only separate, single channels while the more remote farm offers four contiguous channels.

Medium	Throughput	Internet?	Deployed?
LoRa (SF:12)	1.17 kbps	No	**Yes**
DSL	1 Mbps [30]	Yes	No
Satellite	3 Mbps [16]	Yes	No
4G LTE	5.4 Mbps [74]	Yes	**Yes**
TVWS (1x6MHz)	10 Mbps [31]	No	**Yes**
TVWS (4x6MHz)	186 Mbps [31]	No	No
Fiber-optic	1Gbps	Yes	**Yes**

Table 6: Rural uplink throughputs. SF = Spreading Factor.

B.3 Cloud Backup Bottleneck Analysis

The Comosum distributed cloud affords elastic compute and storage power. Practically, we transfer data not only to leverage more abundant compute resources for compute-intensive tasks in the cloud, but also to store the datasets for future retrospective analysis. Assuming each networking media (whether routing in the field or at the gateway) as an unreliable bottleneck in the data transfer, we compare the networking media/latency trade-offs in uploading each application's data.
Table 4 illustrates the expected latencies of different media for a given application. For instance, while a fiber-optic link in an urban farm would transmit CowsOnFitbits' 17MB of data in less than a second, the most popular Internet service in rural locations (DSL) would require three minutes. In another instance, while a research farm with one TVWS channel would route the data within 14 seconds at the edge, the four channels in the rural farm would transmit the same dataset in sub-second time.
In sum, by comparing urban and rural settings, this analysis shows the nuanced networking and storage strategies for DA applications in a distributed cloud setting. This motivates future research avenues in Comosum application migrations either as the network fails or edge resources deplete.

Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing

Giovanni Bartolomeo[♮] *Mehdi Yosofie* *Simon Bäurle* *Oliver Haluszczynski*
Nitinder Mohan[♮] *Jörg Ott*
Technical University of Munich, Germany

{firstname.lastname@tum.de}

[♮] *Corresponding Authors*

Abstract

Edge computing seeks to enable applications with strict latency requirements by utilizing resources deployed in diverse, dynamic, and possibly constrained environments closer to the users. Existing state-of-the-art orchestration frameworks (e.g. Kubernetes) perform poorly at the edge since they were designed for reliable, low latency, high bandwidth cloud environments. We present Oakestra, a hierarchical, lightweight, flexible, and scalable orchestration framework for edge computing. Through its novel federated three-tier resource management, delegated task scheduling, and semantic overlay networking, Oakestra can flexibly consolidate multiple infrastructure providers and support applications over dynamic variations at the edge. Our comprehensive evaluation against the state-of-the-art demonstrates the significant benefits of Oakestra as it achieves approximately tenfold reduction in resource usage through reduced management overhead and 10% application performance improvement due to lightweight operation over constrained hardware.

1 Introduction

Within almost a decade since its inception, edge computing has found a wide range of use cases in industry and research, especially for supporting latency-critical services like AR/VR [24], live video analytics [14], etc. [46]. However, despite significant interest, there have only been a handful of real-world demonstrations of edge so far [49]. Reasons for this are manifold and may include, on the technical side, the following. Firstly, resources at the edge are far less capable and more heterogeneous than datacenters [61] – usually of smaller form factor with specialized hardware, e.g., Intel NUCs [36], Coral AI board [20]), Jetson Xavier [50], Raspberry Pis, etc. Many such devices are designed to be deployed in proximity to the users utilizing unreliable (wireless) networks with limited bandwidth and high latency as primary communication mediums [64]. Moreover, the benefits of edge [47,68] are only apparent with the dense availability of computing resources – requiring significant investment and planning [21,46].

Secondly, the majority of popularly used orchestration frameworks, e.g., Kubernetes [34], K3s [29], KubeFed [35], etc., are off-shoot branches of solutions that were inherently designed to perform well in managed datacenter networks. Such frameworks make strong assumptions about the underlying infrastructure's (especially the network's) consistent reliability and reachability, which does not necessarily hold at the edge where resources are more dispersed. For example, recent investigations into Kubernetes' operations uncovered that its reliance on maintaining strong consistency in the datastore via etcd along with its limited scalability results in severe availability and efficiency issues in edge-like environments [37]. Moreover, the core components of such frameworks incorporate many heavyweight operations – limiting their use on constrained hardware. Furthermore, almost none of the existing solutions can currently support the edge's heterogeneity in hardware, networking, and resource availability.

In this work, we present Oakestra, a flexible, hierarchical orchestration framework that overcomes the many challenges just mentioned. Conceptually, Oakestra allows multiple operators over vast geographical regions to contribute their resources to a federated infrastructure – reducing the investment to achieve a dense computing fabric at the edge. Furthermore, Oakestra's implementation is lightweight and extensible, allowing it to manage effectively constrained and heterogeneous edge infrastructures. Specifically, our contributions are:

(1) We consolidate edge infrastructures in a logical three-tier *hierarchy*. With a root orchestrator managing many resource clusters, each controlled by a cluster orchestrator, we enable infrastructure federation. The cluster orchestrator exercises *local* fine-grained control but only sends aggregated cluster usage statistics to the root (§3). By design, Oakestra hides the internal infrastructure details within each cluster, allowing many providers to participate without exposing internal configurations. Application providers can deploy services at the edge by specifying high-level constraints (hardware, latency, geography) at the root. Oakestra uses a *delegated scheduling mechanism* that decouples the task placement by only making coarse-grained cluster choices at the root and

leaving fine-grained resource placement to the clusters.

(2) We design a novel semantic overlay network that transparently enables edge-oriented load balancing policies (e.g., connect to the closest instance), directly addressable using semantically-capable IPv4 addresses and hostnames (§3.4). This supports the portability of cloud-native applications, ensuring flexibility for application developers to optimize the service-to-service interactions at the edge. The overlay also allows `Oakestra` to dynamically adjust communication endpoints in response to infrastructure changes, e.g., migrations, failures, etc., ensuring uninterrupted service interactions.

(3) `Oakestra`'s lightweight and modular implementation is compatible with most popular cloud technologies and allows developers to extend internal components, e.g., schedulers, without much development overhead (§5). Our extensive evaluation in both high-performance computing and edge infrastructures demonstrates `Oakestra`'s capabilities as it consistently (and significantly) outperforms the popular production frameworks (e.g., Kubernetes and its derivatives). Our results show up to 10× lower CPU overhead, 60% reduction in service deployment time, and 10% application performance improvement. Under heavy loads, `Oakestra` reduces resource utilization by ≈ 20% compared to its closest competitor, K3s. `Oakestra` is an open-source project (`https://www.oakestra.io/`), and all its components are available at `https://github.com/oakestra`.

2 Background and Related Work

Kubernetes [34] has emerged as the most popular orchestration system in production, used by ≈ 59% of all respondents in a recent survey [23], and has been touted by many as the primary solution for edge computing. It decouples the execution runtime of the applications (*nodes*) from the global cluster decisions (control plane). Its smallest deployable units of computing are the Pods, which are a group of containerized services. The *nodes* embed the execution runtime of the pods, as well as the networking e monitoring components of the platform. The *control plane* exposes the APIs for developers and external tools, monitors and synchronizes the nodes, and reacts to cluster events, such as deployments, scaling, and failures. Kubernetes is designed for datacenter environments, and it assumes the nodes to be high-end managed resources interconnected by reliable low-latency networks. The platform guarantees strong consistency of the cluster status and resources in replicated control plane setup via the distributed key-value store called etcd. Recent studies have found that the strong consistency requirements of `etcd` are its primary limitation when ported to heterogeneous and diverse edge infrastructures. This has a noticeable impact on scalability when it comes to constrained resources that can slow down the entire infrastructure [37]. Network partitioning and multi-clustering still remain critical even in Kubernetes federation [35] as shown in [45]. In particular, distributed geographical areas lack cooperation and awareness of the remaining infrastructure. The inter-cluster communication then requires additional tools like Submariner [9] that requires global state transfer synchronization and prevents scalability. Lightweight distributions of Kubernetes such as KubeEdge [19], K3s [29], and Microk8s [43] either inherit the strong assumptions of kubernetes [15] or are meant to perform better on small scale clusters as later shown in our evaluation. In general, while extending Kubernetes or rearchitecting its components is a viable option, we argue that the effort for largely redesigning numerous of the core components would be substantial and instead re-formulate some of its base assumptions. Therefore, `Oakestra` pursues a different approach, built ground-up with the edge requirements in mind, it offers a familiar environment for current Kubernetes developers while providing flexibility to exploit the proximity to their clients and geo-distributed multi-owner infrastructure deployment. `Oakestra` does not aim at superseding the feature set of Kubernetes but rather fills the gap identified in those contexts where Kubernetes does not fit. Our ongoing work explores the integration of Kubernetes-based cloud clusters.

In the literature, we can also find other systems that have explored effective edge orchestration natively. CloudPath [48] envisions multi-tier on-path computing for deploying stateless functions closer to the clients. HeteroEdge [69] or SpanEdge [58] cater specifically towards streaming applications, FogLamp [65] focuses on data management and VirtualEdge [42] only considers edge servers within cellular networks. From the task scheduling perspective, we might relate to different hierarchical scheduler approaches that distribute tasks on a cloud-edge continuum [12, 18, 28, 38]. However, while these solutions focus on service scheduling, in our work, we must integrate the scheduling problem in a comprehensive orchestration framework offering both service and resource management. The work closest to ours is OneEdge [60], as it offers a hybrid two-tier control plane for managing geo-distributed edge infrastructures. However, we consider `Oakestra` to be a superset of OneEdge as the former is a general-purpose modular framework that allows developers to express geographic (and other) management constraints as scheduler *plugins*. We demonstrate such extensibility of `Oakestra` through an LDP scheduler plugin (§3.2) that optimizes on geographical and latency constraints – similar to OneEdge. Therefore, integration remains a possibility which we leave out for future work.

3 `Oakestra` Overview

Previous research has shown that both service deployment and resource management in distributed edge infrastructures are non-trivial problems, primarily due to the heterogeneity and dynamicity of the environment [16, 41, 66]. Simultaneously, the application providers are likely to deploy multiple

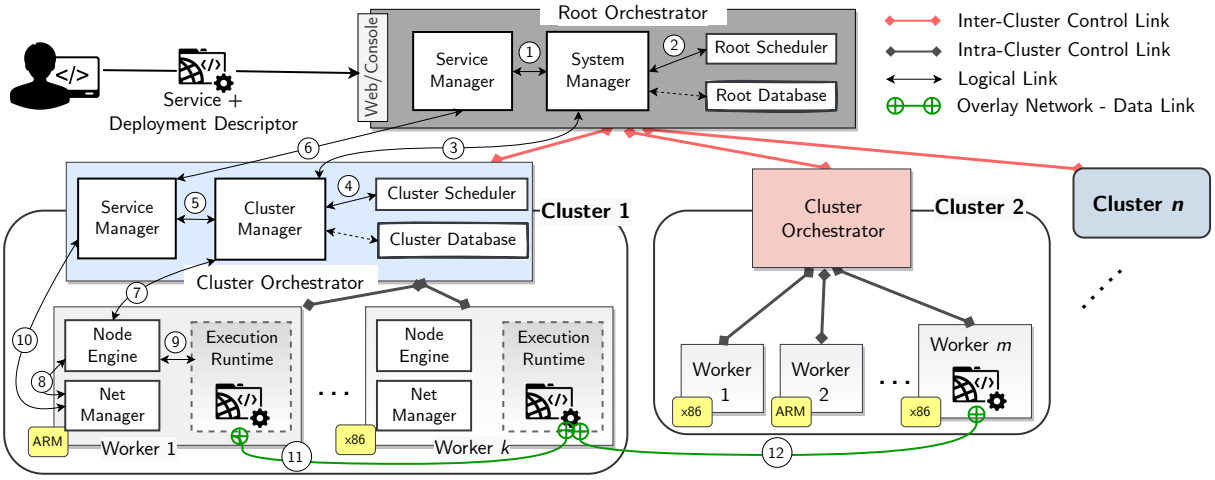


Figure 1: Oakestra Architecture and Workflow.

instances of their services across different edge clusters to have them close to their clients, both in terms of geographic distance and network latency [59]. Additionally, solutions at the edge must remain logically compatible with prevalent cloud techniques such that both existing and novel applications can coexist in edge and cloud realms, thereby providing scalability and flexibility. These and other unique operational viewpoints at the edge impose several design challenges for Oakestra, which we enlist below.

(1) Oakestra must support the infrastructure-at-scale – allowing scaling from thousands to millions of distributed nodes without management overheads. The framework should support a federated heterogeneous infrastructure deployed across geography and controlled by one or more administrative entities and topologies. Furthermore, the framework should allow (a) developers to utilize edge resources regardless of ownership and (b) infrastructure providers to retain management control over their resources.

(2) Extending the infrastructure to the edge of the network requires applications to be able to exploit the proximity with clients. The platform must envision a way for inter-connected microservices to seamlessly communicate and balance the traffic with nearby instances. Therefore, the framework must allow developers to describe the application's requirements with fine-grained SLA primitives (such as specialized hardware requirements, geographical placement, etc.), which must be respected throughout the application lifecycle.

(3) The orchestrator must consider the most up-to-date constraints of edge servers and must adapt to changes in conditions without impacting the applications. Each edge device can contribute with diversified hypervisors, drivers, hardware availability, network, and capacity. The system must abstract the management complexity and autonomously find a compatible node for the deployment issued by the developer.

3.1 System Architecture

Oakestra is a hierarchical orchestration framework for enabling running edge computing applications on heterogeneous resources (Figure 1). Instead of the flat management (inherent to most orchestration solutions [10, 19, 29, 31, 35]), Oakestra organizes the infrastructure into distinct *clusters* (see clusters 1 and 2). We leave the definition of "cluster" purposefully abstract and up to the liking of operators since Oakestra allows multiple edge operators (e.g., ISPs, cloud operators, etc.) to contribute their local deployments towards a shared infrastructure as separate clusters with independent administrative control. Each individual provider then deploys several clusters to segregate its resources, e.g., geographically. The fine-grained control of resources (*workers*) within a cluster is administered by the *cluster orchestrator* (operated by the provider), while *root orchestrator* coarsely controls the global infrastructure. Oakestra can also mimic the single master frameworks (like Kubernetes) with both root and cluster orchestrators deployed on the same machine, albeit with significant performance benefits to the state-of-the-art (§5). The resource and service management responsibilities are separated into independent components, `system manager` and `service manager` (§3.2). We carefully design and implement Oakestra as a modular and extensible framework – allowing the possibility to swap technologies and/or add new features as the requirements of edge computing evolve in the future (§5). Oakestra comprises of three main entities – *root orchestrator*, *cluster orchestrator*, and *worker*.

The **Root Orchestrator** is Oakestra's centralized control plane (analogous to Kubernetes's "control-plane" [10]) and is responsible for managing resource clusters. However, as we explain in §3.2, the root only provides (i) coarse high-level control and (ii) interactions across multiple clusters, as fine-grained control is retained within the cluster boundary. Regardless, we envision the root to be deployed on a machine

reachable from all clusters (in a widescale deployment, e.g., in the cloud). While the root orchestrator may appear to be a central point-of-failure initially, the context separation into fine- and coarse-grained management responsibilities across hierarchy allows Oakestra to continue its operations if the root fails and restarts (see *Fault Tolerance in §3.5)*.

To deploy applications, developers submit the code along with an SLA descriptor to the system manager. The SLA includes high-level operational requirements and constraints for service execution at the edge, e.g., virtualization, required hardware, geolocation, etc. (see §3.2). The system manager notifies the service manager of the new deployment request (step ①), and contacts the root scheduler (step ②) to calculate a priority list of clusters (based on aggregate information) to deploy the application. As such, Oakestra follows a multi-step *delegated service scheduling* approach as the root offloads the fine-grained scheduling operation to selected clusters schedulers (details in §3.2). The system manager is also responsible for registering new clusters and coordinating the control information.

The **Cluster Orchestrator** is a logical twin of the root but with management responsibility restricted to resources within the local cluster. An infrastructure provider registering its resources as a cluster with the root assigns the orchestrator role to a machine that is ideally reachable by all workers. The cluster manager periodically updates the root with *aggregated* statistics of overall cluster utilization and health/QoS of the deployed services (step ⑤, and ⑥) via HTTPS-based *inter-cluster control link* in a *push-based* manner (implementation details in §5). Note that the cluster orchestrator withholds minute information and retains majority administrative control of its member workers. For example, if the cluster orchestrator receives a delegated scheduling request from the root (step ③ and ④), it calculates the optimal resource selection considering the up-to-date availability, utilization, and capability reported by the attached workers. The Oakestra scheduler is designed to be modular and supports several different scheduling algorithms as language-agnostic *plugins*.

Worker Nodes are edge servers in clusters responsible for executing services. Each worker has a distinct capacity and capability, e.g., CPU, GPU, disk, RAM, etc., which it reports to the local cluster orchestrator at registration. If a worker's capacity and capability match the service's SLA constraints, the cluster orchestrator instructs the worker's NodeEngine to deploy the service ⑦, triggering a runtime (and network) instantiation ⑧ and service execution ⑨. Each worker periodically reports its utilization, health of operational services, and (potential) SLA default alarms to its cluster orchestrator via an MQTT-based *intra-cluster control link*. Note that for interconnecting microservices deployed across clusters, Oakestra does not require workers to have public IP addresses as the net manager natively supports both direct ⑪ and tunneled ⑫ communication (see §3.4 for details).

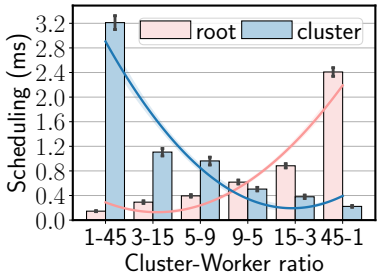


Figure 2: Scheduling time for different cluster sizes.

Why hierarchical orchestration? To understand if a hierarchical management design is inherently better for scalability at the edge, We created an experimental setup with 45 worker node VMs in a compute cluster and configured the orchestration with increasingly different cluster sizes. Starting from a single cluster with 45 nodes, we gradually increased the number of clusters up to 45 with only one worker. For each split, we performed 100 deployments and recorded the time the root and cluster scheduler took for the scheduling decision (see fig. 2). Note that the corner cases of the configuration, i.e., one cluster with 45 workers ($1-45$) and 45 clusters with one worker each ($45-1$), represent the flat orchestration design used in majority state-of-the-art, e.g., in Kubernetes-based solutions. The results indicate that decoupled orchestration reduces scheduling time, primarily since the infrastructure search space is a smaller subset. Optimal performance is achieved when the workers are somewhat balanced across multiple clusters in the hierarchy, and the minimum is around nine clusters with five workers. It is apparent (from the $3-15$ split) that using a two-layer hierarchy can be advantageous, especially for wide-scale infrastructure.

3.2 Resource and Service Management

Resource Management. As discussed in §3.1, edge resources in Oakestra are deployed in distinct clusters, with $R^i = \{R^i_1, R^i_2, \ldots, R^i_n, R^i_{CO}\}$ resources in the ith cluster. Here, R^i_{CO} is the cluster orchestrator of i-th cluster. Each resource R^i_n periodically *pushes* its current utilization (U^i_n) and other characteristics (e.g. location) to R^i_{CO} with update frequency $\lambda(R^i_n)$ over the intra-cluster communication channel. At each update, R^i_{CO} calculates the available capacity of R^i_n by correlating U^i_n to the maximum capacity C^i_n reported at registration. $\lambda(R^i_n)$ can be dynamically tuned for each R^i_n to balance between network overhead and the freshness of the information. We leave the exploration and impact of $\lambda(R^i_n)$ algorithms and the use of techniques such as AoI [62] to future work. Similar to intra-cluster, the update messages over inter-cluster links are also push-based. Each cluster orchestrator periodically sends the aggregate distribution of *available* current capacity, i.e. $\cup(A^i) = \langle \sum(A^i), \mu(A^i), \sigma(A^i) \rangle$ where $A^i = \{A^i_1, A^i_2, \ldots, A^i_n\}$ to the root. The aggregation allows different operators to (i)

```
constraints: [{
    microservice_id: {type: number},
    properties: [{
        memory: {type: integer},
        vcpus: {type: integer},
        vgpus: {type: integer},
        vtpus: {type: integer},
        bandwidth_in: {type: integer},
        latency: {type: number},
        area: {type: string},
        location: {type: string},
        threshold: {type: number},
        rigidness: {type: number},
        convergence_time: {type: integer},
        virtualization: {type: string},
        ... }],
    ...}]
```

Schema 1: Service Requirement Descriptor.

participate in the federated infrastructure while obscuring the minute details of their resources and (ii) freely scale up/down their cluster density without involving the root.

Service Deployment & Scheduling. Developers can deploy their (standalone or multi-microservice) applications by specifying high-level QoS requirements within the SLA description. The deployment descriptor used for submitting applications to `Oakestra` is composed of (i) a description of each microservice of the application (name, namespace, image, etc.) along with communication interlinks, and (ii) a service level agreement (SLA) describing the constraints that the platform must respect for each one of them. Schema 1 shows a fragment of the high-level SLA description supported by our framework. In addition to operational requirements already prevalent in cloud environments, such as processing performance, networking requirements, virtualization needs, etc., the schema allows developers to specify edge-specific restrictions, e.g., geographical location, specialized hardware, etc. Additionally, developers can fine-tune the precision of scheduling heuristics by enforcing *convergence time* and *decision rigidness* metrics. Convergence time specifies the maximum allowed time within which the scheduler should find the suitable edge server that supports the SLA requirements of the service, and rigidness defines the sensitivity for re-triggering service scheduling in case the selected resource violates the SLA (due to environment/infrastructure changes).

As described earlier, `Oakestra` follows a two-step *delegated scheduling* mechanism. Specifically, upon receiving a service deployment request from the developer, the `root scheduler` matches the SLA constraints to the current capacity of each cluster and calculates a priority list of best-fit clusters based on the latest aggregate cluster usage distribution. The root then offloads the deployment request (including SLA and the service) iteratively to cluster orchestrator(s) with decreasing priority. Upon receiving the request, the `cluster scheduler` calculates the optimal service placement within its cluster, leveraging the available schedulers (see §3.2.1). Note that we design `Oakestra`'s scheduling logic to be *language-agnostic* – allowing developers/researchers to implement custom algorithms as *plugins*.

Algorithm 1: Resource-Only Match

Input: A_n : Information about worker n.
$Q_{\tau_{p,i}}$: Requirements of i-th task of p-th service.
$f(A_n, Q_{\tau_{p,i}})$: Resource selection strategy.
Output: Best worker W to run $\tau_{p,i}$.

// Resource selection strategy examples:
// $f(A_n, Q_{\tau_{p,i}}) = *arg\,max_n \big[(A_n^{cpu} - Q_{\tau_{p,i}}^{cpu}) + (A_n^{mem} - Q_{\tau_{p,i}}^{mem})$
// $\wedge\, Q_{\tau_{p,i}}^{virt} \in A_n^{virt}\big]$
// $f(A_n, Q_{\tau_{p,i}}) = *first_n \big[Q_{\tau_{p,i}}^{cpu} \le A_n^{cpu} \wedge Q_{\tau_{p,i}}^{mem} \le A_n^{mem}$
// $\wedge\, Q_{\tau_{p,i}}^{virt} \in A_n^{virt}\big]$
1 $W \leftarrow f(A_n, Q_{\tau_{p,i}})$
2 **return** W

`Oakestra`'s delegated scheduling significantly reduces the search space of the multi-objective task placement problem by considering a subset of resources at each step. In case all microservices of an application cannot be placed within the same cluster, the root scheduler iteratively requests other clusters in the priority list for pending deployment(s). The worker `node engine` also keeps track of the deployed services through their lifecycle (see §3.3). In case of *failures* (resource – if the last update from a worker exceeds a threshold; service – if a worker raises an alarm), the cluster orchestrator marks all affected services as failed and attempts to re-deploy them on another suitable resource within the same cluster. If unsuccessful, the rescheduling request is propagated to the root for system-wide scheduling. Similarly, the cluster orchestrator can trigger *re-deployments* if it observes any SLA violations (exceeding specified rigidity).

3.2.1 Service Schedulers in `Oakestra`

Let $S = \{s_1, s_2, \ldots, s_{|S|}\}$ denote the set of services requested to be deployed by the developers at the root. Each service $s_p \in S$ can be composed of n individual microservices or *tasks*, i.e. $s_p = \{\tau_{p,1}, \tau_{p,2}, \ldots, \tau_{p,n}\}$ where $\tau_{p,i}$ denotes i-th task of p-th service. Each task $\tau_{p,i}$ requires a certain capacity (CPU, GPU, memory), denoted by $Q_{\tau_{p,i}}$. Other considerations like geographical location or virtualization technology, specified by the developer in the SLA, are also part of $Q_{\tau_{p,i}}$. The task of the scheduling components (in both root and cluster) is to find a suitable resource in the infrastructure that supports the requirements in $Q_{\tau_{p,i}}$. In this work, we propose and incorporate two different scheduling approaches.

(1) Resource-Only Match (ROM): As the name suggests, in ROM, the cluster scheduler finds a suitable resource that satisfies the service's capacity requirements (see Algorithm 1). The scheduling approach is analogous to greedy-fit and knapsack-based solutions popularly used for placing VMs on cloud servers in datacenters [63].

(2) Latency & Distance Aware Placement (LDP): LDP (shown in Algorithm 2) builds on the ROM scheduler but additionally considers latency and geographical distance con-

Algorithm 2: Latency & Distance Aware Placement

Input: A_n : Information about worker n.
$Q_{\tau_{p,i}}$: Requirements of i-th task of p-th service.
Output: Best workers W to run $\tau_{p,i}$.

$W \leftarrow \{n \in [1,|A|] \,|\, A_n^{cpu} \geq Q_{\tau_{p,i}}^{cpu} \wedge A_n^{mem} \geq Q_{\tau_{p,i}}^{mem} \wedge Q_{\tau_{p,i}}^{virt} \in A_n^{virt}\}$
if $|Q_{\tau_{p,i}}^{s2s}| \geq 1$ **then**
  for Q_j *in* $Q_{\tau_{p,i}}^{s2s}$ **do**
    $t \leftarrow Q_j^{trg}$
    $W \leftarrow \{n \in W \,|\, dist_{gc}(A_n^{geo}, A_t^{geo}) \leq Q_j^{geo\_thr} \wedge dist_{euc}(A_n^{viv}, A_t^{viv}) \leq Q_j^{viv\_thr}\}$
  end
end
if $|Q_{\tau_{p,i}}^{s2u}| \geq 1$ **then**
  for Q_k *in* $Q_{\tau_{p,i}}^{s2u}$ **do**
    $u \leftarrow Q_k^{lat\_trg}$
    $rtts \leftarrow \{rtt_{i,u} \,|\, i \in rnd(W), rtt_{i,u} = ping(i,u)\}$
    $vivaldiNet \leftarrow \{A_n^{viv} \,|\, n \in [1,|A|]\}$
    $U \leftarrow trilateration(rtts, vivaldiNet)$
    $W \leftarrow \{n \in W \,|\, dist_{gc}(A_n^{geo}, Q_k^{geo\_trg}) \leq Q_k^{geo\_thr} \wedge dist_{euc}(A_n^{viv}, U) \leq Q_k^{lat\_thr}\}$
  end
end
return W

straints for service placement. Since edge applications can be composed of multiple microservices that can either interact with each other (in a chain-like fashion) or directly with end users/devices, we allow the application provider to specify constraints for both service-to-service (S2S) and service-to-user (S2U) links. The root scheduler first filters unsuitable clusters by comparing their resource constraints along with approximate geographical operation zones to the SLA requirements. Within each cluster, the algorithm first creates a list of candidate workers that satisfy the resource constraints. Then, for all S2S constraints $Q_{\tau_{p,i}}^{s2s}$, the algorithm filters out workers that exceed the specified distance $Q_j^{geo\_thr}$ and latency thresholds $Q_j^{viv\_thr}$ to the target service $t = Q_j^{trg}$. LDP estimates geographic distance as the great circle distance ($dist_{gc}$) between the geographic location of worker n (A_n^{geo}) and the location of the target service A_t^{geo}. The approximated latency is the Euclidean distance ($dist_{euc}$) between the location of worker n (A_n^{viv}) and the location of the target service A_t^{viv} in the Vivaldi network [25]. Vivaldi is a network coordinate system embedding networked nodes into a d-dimensional coordinate system such that the Euclidean distance of two nodes approximates their round-trip time. If the developer has specified any S2U constraints $Q_{\tau_{p,i}}^{s2u}$, LDP measures the round-trip times ($rtts$) to the target as $Q_k^{lat\_trg}$ from a set of random workers in the cluster ($i \in rnd(W)$). The measurements approximate the user's position within the Vivaldi network via trilateration. Following that, LDP filters out workers that exceed the distance threshold $Q_k^{geo\_thr}$ to $Q_k^{geo\_trg}$ or the latency threshold $Q_k^{lat\_thr}$ to the approximated user position U.

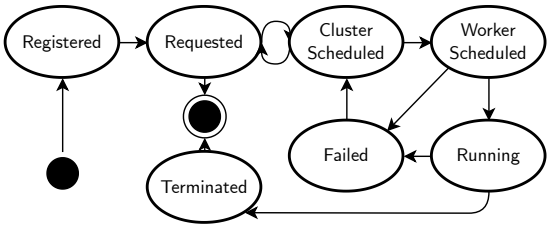


Figure 3: Lifecycle of a service's instance.

3.3 Application Lifecycle

In `Oakestra` an application is composed of multiple services. Each service, in turn, is composed of multiple instances. `Oakestra` keeps track of the instances deployed in the platform through a lifecycle state machine (fig. 3). At any point in time, an instance can have one of the following states.

`Registered`: The developer has submitted an application and its SLA (shown in fig. 1). The service manager in the root saved the service SLA and generated the instance metadata.

`Requested`: After receiving the deployment command, the instance is sent to the root scheduler, and the system manager awaits a suitable cluster. When there is no cluster able to guarantee the SLA, the deployment cannot be carried out.

`Cluster Scheduled`: The root scheduler designated a suitable cluster. The instance SLA is sent to the corresponding cluster orchestrator. The cluster orchestrator is now waiting for the cluster's scheduler decision. If no worker node is capable of hosting the instance, the request is sent back to the root in `Requested` status.

`Worker Scheduled`: The cluster scheduler finds a suitable worker node. The instance binaries (or image download details) are fetched from the root and transferred to the selected worker node while the ports for networking are allocated.

`Running`: The instance is operational as it satisfies the SLA constraints. The worker periodically tracks the current QoS and relays it to the cluster orchestrator along with current utilization in periodic *heartbeat* update messages.

`Terminated`: The service is no longer operational due to an explicit `terminate` command from the cluster orchestrator (issued by the developer at the root). Alternatively, the service gracefully finishes its execution and terminates.

`Failed`: The service execution has stopped with unexpected exit status. A service "fails" if the worker node explicitly reports this state to the cluster orchestrator as a failure alarm or if the node fails. A resource is considered "failed" if it has not sent a *heartbeat* for longer than a pre-defined threshold.

3.4 Service Communication

Supporting intra-service (and service-to-user) networking at the edge can be challenging since (i) infrastructures are susceptible to dynamic changes, (ii) application deployment is

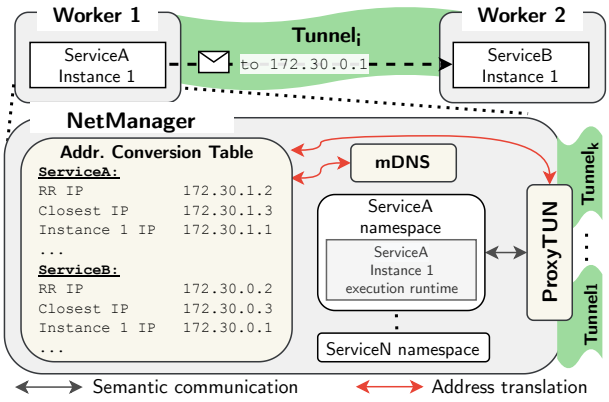


Figure 4: Service communication across edge servers.

fluid to remain close to clients [67], and (iii) several service instances can coexist simultaneously to achieve broader coverage, therefore addressing and balancing must dynamically adapt to the service placement. Moreover, it is impractical to presume that edge servers from multiple participating infrastructure operators can interact over a common/public network – an assumption implicit in the majority of existing solutions [29, 34, 43]. `Oakestra` includes a component called `NetManager` that enables: (a) dynamic routing policies transparently enforced via semantic service addressing to support load balancing catering to edge environments (§3.4.1) and (b) transport layer packet tunneling to interconnect services operating on resources with limited accessibility (§3.4.2).

`Oakestra` separates the bulk of data-plane complexity from the control-plane operations at the worker level. Particularly, the framework utilizes the branches in the multi-tier edge hierarchy only for core control-plane information propagation (e.g., routing updates, service/hardware utilization, and failures, etc.) and designates data-plane communication management to leaf workers using the `NetManager` component. Figure 4 shows the cross-section of `NetManager` with communication between services A and B deployed on workers 1 and 2, respectively. The `proxyTUN` actively maintains and dynamically adjusts endpoints of the tunneled connections to adapt to infrastructure changes, ensuring uninterrupted communication between services. Arguably, the proposed approach resembles some sidecar proxy solutions like Istio [3]. While the set of proposed functionalities might be similar, the `NetManager` does not need to deploy a sidecar along with each deployed application. While the design of solutions like Istio fits the abundance of resources in the cloud context, we propose a lightweight approach with a worker-level proxy used by all the applications, like the native Kubernetes network but featuring additional balancing flexibility and site-to-site tunneling out-of-the-box.

3.4.1 Service Naming and Addressing

Drawing inspiration from semantic routing [39], a `serviceIP` addresses all the instances of the service according to different balancing policies (e.g. *round-robin*, *closest*, etc.). For example, in fig. 4, worker 1 maintains the ServiceA's `closest` and `round-robin` `serviceIP`s – allowing services to select balanced instances through IP addresses. Note that the serviceIP is different from the host IP address as the former is ephemeral and addresses service instances (similar to ClusterIP in Kubernetes [34]). When receiving a packet to an address belonging to the load-balanced `serviceIP`, the `ProxyTUN` uses the conversion table to get the corresponding `Instance IP`. If no conversion entry is found, the `NetManager` registers an interest in that route and enquires the cluster component (see (10) in fig. 1). If the cluster orchestrator does not contain the information, the interest registration is propagated to the root. The root either knows the service route or the service does not exist at all, and no route can be propagated. The interest registration also allows the worker to receive future updates regarding the route. When a route is not used, the conversion entry is erased, and the interest is deregistered.

If properly configured, services can use DNS-based naming schemes which resolve to a *serviceIP* using a **mDNS** [57]. We envision a service naming schema that reflects the hierarchy `<instance_number>.<routing_policy>.<service_name>.<service_namespace>.<app_name>.<app_namespace>`. The `app name` and `namespace` portion of the domain is provided by the developer to uniquely address the application (e.g. videoAnalytics.org). The `service name` and `namespace` address different microservices in the application (e.g. ServiceA.default, ServiceB.default, etc.) while the `instance` number uniquely distinguishes individual service replicas. In case the developer does not care about connecting to a specific instance of the service, `<instance_number>` can be set to "any". The unique aspect of the proposed service naming is the `routing_policy`, as it allows developers to offload connection endpoint selection to `Oakestra` based on the current deployment state. For example, the "closest" policy resolves to serviceIP address representing the nearest service instance while "round-robin" balances connection load across multiple instances. Following the example of fig. 4, suppose *ServiceA* must send a request to the closest instance of *ServiceB*. The request can either be addressed using the semantic IPv4 address representing the *closest* routing policy, therefore `172.30.0.3`, or the name `any.closest.ServiceB.default.videoAnalytics.org`. The *ProxyTUN* component will convert the given semantic address to and *instance address* that represents the chosen instance accordingly to the balancing policy. It will then proceed tunneling the packet towards the destination.

The reachability of the services managed by `Oakestra` from external users can be achieved with standard DNS and API getaway solutions. For future extensions of this work, we are investigating techniques that can support client mobility and dynamicity of the endpoints for first-mile computation and fast handovers. In fact, traditional cloud solutions in edge contexts lead to frequent DNS resolutions and a lack of sup-

port for the discovery of services in close user proximity.

3.4.2 Connection proxying and tunneling

Oakestra enables inter-service communication across workers in different clusters with limited available ports (e.g. behind firewalls) through UDP tunneling. We can again refer to the example shown in fig. 4, where *ServiceA* on *worker 1* needs to communicate with *ServiceB* on *worker 2*. Every packet sent from *ServiceA* to *ServiceB* is handled by the **proxyTun** attached to a virtual bridge in worker 1. The *proxyTUN* component resolves the serviceIP of the destination *ServiceB*, selecting the *Instance IP* accordingly to the balancing policy enforced. If no resolution entry is found, the worker node subscribes to the updates regarding this route to the cluster orchestrator. The resolution entries for a service include a list of the available service instances, their respective IP addresses assigned from each local worker's subnetwork (Instance IPs), and each one of the destination worker's address and port that should be used for the tunneling. The tunneling is performed at L4 using UDP. The L4 implementation allows to transparently support all transport protocols (TCP/UDP/QUIC) out of the box. The only requirement in order to support nodes behind NATs and firewalls is to provide a port for the site-to-site tunneling. An open tunnel (i, j) connecting node i to node j for performance reasons is recycled for all the traffic of all the services communicating with instances belonging to those workers. Therefore, each node only has one ingress tunnel and k egress tunnels, one for each of the workers it's currently exchanging traffic with. To support router configurations without any open incoming ports we envision, in the future, allowing service's connections to transit via the cluster orchestrator. In this case, the *service manager* acts as a VPN server that tunnels the traffic between worker nodes.

If the *NetManager* cannot forward a message (e.g., due to out-of-date information), the route is immediately deleted, and a route refresh is performed. This mechanism avoids imposing a strong consistency requirement on the worker's caches. While this approach might miss the best balancing option when a route update is still propagating asynchronously, it reduces the synchronization effort and, thus, overhead.

We note that by assigning the majority of networking complexity to the worker node, Oakestra dramatically reduces the overhead of orchestration machines. The *service manager* sends only the most relevant routes for each service (according to an internal worker-wise priority list) to increase the scalability and avoid further congestion on the workers. The inherent decentralized networking design of Oakestra is also tolerant towards infrastructure failures. For instance, coupled microservices will continue to communicate with each other even if the cluster (and root) orchestrator becomes unavailable – as long as the host worker node is operational and the endpoints do not migrate (to a different cluster).

3.5 Fault Tolerance.

We now explore the different possible failure cases and how Oakestra manages them. While Oakestra decouples the two-tier *master-worker* orchestration design, prevalent in popular frameworks, in a hierarchical three-tier infrastructure, it is still dependent on the *root*, which may appear as a central point of failure. In case the *root orchestrator* fails, the platform is unable to register new applications as well as schedule new instances to new clusters. Applications that are already part of a cluster can locally replicate, scale, and migrate. The networking is partially affected by root failure since existing tunnels and communication will continue to work, but new inter-cluster links require the root network component for the setup process. Each cluster's aggregated information cannot be propagated, but it will be stored until the root is back online. Since the root is likely to be deployed in the cloud to maximize reachability, it is less frequently affected by failures due to hardware issues. Moreover, in such contexts, traditional redundancy and failover mechanisms can potentially be implemented with redundant replicas, synchronized key-value stores, and L7 load balancing to properly route the cluster's traffic to the active root instance.

A *cluster orchestrator* failure does not affect other clusters' activities. As soon as a cluster stops responding, the root marks the applications deployed within that cluster as failed and attempts to replicate the existing workload to new suitable clusters. The worker nodes will not be able to update their status on the failed cluster orchestrator, but the applications will remain operational. The root will then schedule new backup application replicas in the remaining clusters, and the network routes will reactively change toward the newly managed instances. *Worker node* failures may be frequent and expected at the edge; for this reason, the workloads deployed on a failed node are immediately rescheduled. The network automatically adjusts and balances the traffic to the active instances disregarding the unreachable ones. The cluster orchestrator discontinues aggregating failed node's resources from the cluster's pool in further updates.

In addition to the current fault tolerance strategies and assumptions, in future extensions, we envision having multiple replicas of the orchestrator components that coarsely stay in sync with the primary (similar to the multi-master setup in Kubernetes) and/or a leader election strategy to react to control plane failures. Moreover, we intend to explore the orchestration problem in contexts where byzantine behaviors can be expected at both worker and cluster levels.

4 Implementation

We implement Oakestra and its components (fig. 1) with constraints of heterogeneous edge infrastructures in mind. The implementation, spanning 18000 LOC, is modular, extensible, lightweight and open-source [52]. As we show later,

Oakestra can support production-ready applications at the edge much more effectively than the state-of-the-art.

Root and cluster orchestrators are implemented using Python in a similar structure owing to their architectural similarities. The schedulers, system/cluster manager and service manager are implemented as independent micro-services. The database is implemented as two separate instances of mongoDB [5], one for the service manager and one for the system/-cluster manager. The implementation does not force strong consistency in cluster's or root's data structures. They update asynchronously, improving the overall system's scalability – at the cost of occasionally dealing with application rescheduling. The scheduler microservice uses a celery task queue [1] to pull the incoming tasks and find the best placement asynchronously. Scheduling policies are described in Python but are designed to be language-agnostic. The system and cluster managers communicate with the database services and the service manager via REST APIs. The external root APIs for the infrastructure providers and application developers are implemented according to the Open API Specification [7] and authenticated using JWT tokens [4]. Inter-cluster links are RESTful APIs as well while the intra-cluster control links are implemented using MQTT. With a Mosquitto broker [6] hosted at each cluster orchestrator, a worker can subscribe to network routes updates, and publish internal resource consumption and task status. We also provide an Angular web-based frontend attached to the root that can be used by the developers to facilitate (i) creation of applications and describing the services graphically, (ii) monitor each service instance's status and position (fig. 13b), and (iii) scale the services up and down (see Appendix A).

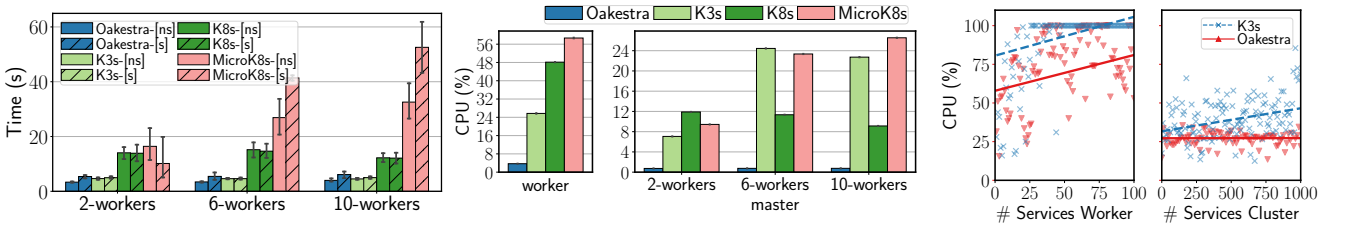
The NodeEngine component as well as the NetManager are implemented as independent services using GoLang to ensure low footprint and maximum support to many execution run-times SDKs. The container's execution runtime is supported with the integration of containerd [2]. Unikernel [40, 44] support is currently under development with QEMU [8] virtualization. The runtime selector is designed to be extended to support even more virtualization technologies in the future, e.g. microVM [11]. The NetManager component uses native Linux virtual network interfaces to create a bridge connected via veth pairs with the TUN interface (namely proxyTUN) and the service's network namespace. The *NetManager* minimizes system context changes and makes extensive use of Goroutines pools to resolve the incoming traffic with high parallelism. The semantic addresses are reserved from a pre-defined 10.30.0.0/16 sub-network. The traffic belonging to the semantic sub-network is forwarded to *proxyTUN*.

5 Evaluation

This section focuses on evaluating Oakestra as compared to the state-of-the-art on edge infrastructures and constrained devices. We use two different testbeds for our evaluation. The *High-Performance Computing (HPC)* testbed is a large, controlled, x86 processor-based cluster, in which we use *S*, *M*, *L*, *XL* VMs with <1,1>, <2,2>, <4,4> and <8,8> <CPU core,RAM (GB)> configurations, respectively. We use this cluster to flexibly spawn resources and emulate a heterogeneous infrastructure. Our *Heterogeneous (HET)* testbed is a local cluster composed of Raspberry Pis [33], Intel NUCs [36], mini-desktops, and Jetson Xavier [50] – representing different edge computing flavors [68]. The HPC cluster is interconnected by 1 Gbps Ethernet, while HET machines connect over Wi-Fi 802.11ac and 1 Gbps Ethernet links. We attempted to compare Oakestra against popular orchestration frameworks. However, despite careful management, KubeFed [35], KubeEdge [19], ioFog [31] and Fog05 [30] experience frequent failures, possibly because they are (i) in early development stages or (ii) not optimized for constrained hardware. Moreover, we could not locate OneEdge's source code [60], which is the only framework architecturally similar to Oakestra. As a result, we compare Oakestra against Kubernetes (K8s) [34] and its two lightweight derivatives, MicroK8s [43] and K3s [29]. All selected frameworks are widely used and have been considered for use with the edge [15, 37]. We use two application workloads, (i) an Nginx web server allowing us to control the operational load dynamically, and (ii) a video analytics application from [13]. The latter is composed of four microservices. The *source* sends a pre-recorded RTP stream [17], *aggregation* stitches and pre-processes each frame, *detection* uses YOLOv3 to detect objects, and *tracking* tracks objects across frames. To remain comparable with the "kube" frameworks, we operate Oakestra in standalone mode, i.e., all workers are deployed within the same cluster. We perform 10 runs for each experiment and clean intermediary files between runs.

5.1 Service Deployment

Figure 5a compares the time taken by each framework to deploy a containerized application on the infrastructure. For this experiment, we configure an *XL* VM as root, an *L* VM as cluster orchestrator in Oakestra (and master for others), and *S* VMs as workers. Oakestra uses the ROM scheduler, which is comparable to the default scheduling policy of the competitors [32]. We increase the cluster size from 2 to 10 workers and measure scheduling overheads by toggling its operation, shown with *s* (with scheduler) and *ns* (no scheduler). MicroK8s performs significantly worse ($\approx 10\times$ slower) than Oakestra, degrading further with increasing infrastructure size. As also noticed in [15], microK8s might easily lead to higher resource usage and generally slower performance. We attribute it to (i) *snap*, which brings extra virtualization overhead, and (ii) microk8s being optimized for single-node deployments. Kubernetes is 2–3$\times$ slower than Oakestra. Its scheduling operation adds almost negligible overhead – this



(a) Deployment time

(b) CPU utilization

Figure 5: Performance comparison for different infrastructure sizes.

Figure 6: CPU usage of worker & cluster orch. in stress (line=median).

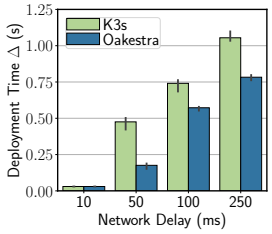


Figure 7: Deployment time with network delay.

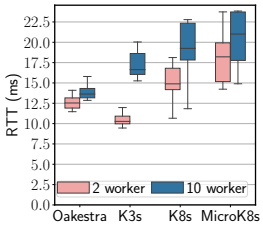


Figure 8: End-to-end latency

result is in line with other recent explorations [15]. One of the reasons why K3s has been generally considered lightweight is its single binary executable that also reduces the internal overhead and synchronization time. This approach, used by `Oakestra` as well, resulted in a comparable deployment time.

Repeating the experiment in our HET testbed, we introduced increasing networking delay using `tc` on the network interface to highlight the overhead due to master-worker synchronization. In fig. 7, we observe that `Oakestra`'s deployment times improve over K3s by a $\approx 20\%$ margin in high delay networks (common for wireless last-mile [22,26,46]). The behavior is similar for lossy networks as `Oakestra` achieved $\approx 50\%$ and 60% deployment time reduction with 20% and 50% losses, respectively (not shown for brevity). The eventual consistent store of `Oakestra` allows the platform to asynchronously manage the deployment with reduced network traffic. As shown later in §5.5, "kube" orchestrators send $\approx 2\times$ more control messages (from both worker and master) on average – hinting at the cause of their degradation to aggravating network conditions. Experiments on the HET cluster typically yielded better results than those on HPC. This is because a more powerful (yet still limited) Raspberry Pi 4 was used as the target device in HET, as opposed to the size *S* worker nodes utilized in HPC.

5.2 Scalability

First, we analyze the idle resource consumption of each framework to estimate the baseline overhead. Lower overhead at the worker indicates a platform's capability to operate on constrained devices. In comparison, lower overhead at the master (at different infrastructure sizes) highlights its ability to handle scale (fig. 5b). We observe that within the chosen competitors, K8s is the one that better handles cluster scalability at the masters, showing no noticeable increase in CPU usage. Meanwhile, K3s and MicroK8s masters show a lower average CPU usage than k8s in two-worker setup but much worse degradation in 6 and 10 workers clusters. At the worker level, k3s is almost 50% faster than K8s. MicroK8s exhibits, again, a higher footprint. We attribute this to the same motivations expressed in §5.1. Due to its asynchronous pub-sub communication and its cluster resource aggregation mechanism, `Oakestra` achieves $\approx 6\times$ and $11\times$ reduction in CPU on the workers and master, respectively. Particularly, an `Oakestra` worker only maintains an MQTT connection to the cluster's broker to periodically report the device resource usage. The orchestrator, subscribed to worker resource topics, only updates the internal database with the latest worker utilization information. Compared to k8s, `Oakestra` maintains a considerably reduced duty cycle while idling.

Since K3s comes closest to `Oakestra` at the worker's level, we perform a stress test comparing both for increasing service deployments. Figure 6 compares the CPU consumption as we increasingly schedule up to 100 Nginx containers on each worker in a 10 node cluster (totaling 1000 containers in the cluster). The left and the right half show the worker and cluster orchestrator (or K3s master) utilization, respectively. `Oakestra` sees negligible overhead performing $\approx$ 10–20% better than K3s – demonstrating its efficacy to support large service volumes. `Oakestra`'s average cluster CPU usage increases by less than 1% during the experiment since each node piggybacks the service status onto the internal resource consumption updates, lowering network usage and processing cycles at the orchestrator. Similarly, `Oakestra` shows significant operational advantages for constrained worker nodes. While K3s exhausted the worker's CPU at $\approx$ 40 services, `Oakestra` deployed the 100 services with a 30% average CPU surplus. Memory utilization showed a similar trend as `Oakestra` achieved $\approx$ 18% and $\approx$ 33% reduction to K3s in worker and master, respectively (not shown).

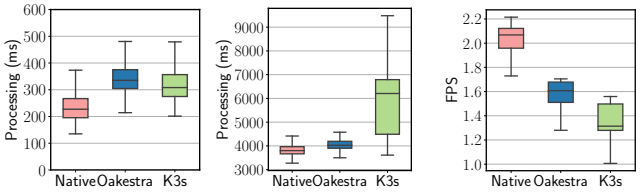
5.3 Networking

We evaluate the round-robin balancing policy of the networking scheme presented in §3.4 against the native balanced *cluster IP* of K3s, K8s, and MicroK8s. First, in a 2-worker setup, we deploy a Python client on the first worker and an Nginx server on the second worker. We then scale the number of workers to 10 and deploy one client and 9 Nginx servers, one on each worker node. The client performs continuous GET requests using a round robin *service IP* on `Oakestra` and a *cluster IP* on the *kube* family with a statically configured round robin policy. Figure 8 shows the average round-trip latency between the client and the closest server. On average, K3s performs better in a 1-to-1 (2 workers) setting (10–20% improvement), while Kubernetes and MicroK8s perform 17% and 30% slower. All competitors' load balancing is significantly worse than `Oakestra` in multiple replica settings, resulting in ≈ 15 up to 35% RTT inflation. The results show the benefits and overhead of the proposed addressing scheme. `Oakestra` performs proxying and site-to-site tunneling for every packet, even in a simple setup with just one client and one server, slightly increasing the overhead even in LAN setups, like the experiment above. On the other hand, this abstraction brings benefits when scaling up the system, introducing minimal additional overhead while balancing with more replicas and outperforming the other systems. In the future, we plan to optimize the 1-to-1 scenario by temporarily disabling the proxy and utilizing VXLAN-based solutions for nodes belonging to the same network.

We evaluated the impact of `Oakestra`'s tunneling on the bandwidth. While the proposed network component is mainly designed to implement semantic addressing, it can also tunnel the traffic between nodes on different networks. For this reason, we test the impact on the bandwidth by comparing it with `WireGuard` [27] – an open-source tunneling solution used by most frameworks. We emulate the network inconsistencies at the edge [46] by gradually increasing the delay between the client and the servers from 10 to 250 ms. Figure 12 compares the time to download a 100 MB file over HTTP using both approaches. We find that, even in high-delay networks, `Oakestra` is always within the competitive range (2-10%) of `WireGuard`. We consider this a promising result given that the proposed networking component performs proxying on top of the "simple" tunneling operations of WireGuard.

5.4 Scheduler Performance

This section shows a preliminary evaluation of ROM and LDP schedulers. Figure 10 depicts both schedulers' performance in a simulated infrastructure with up to 500 edge servers (virtually configured in HPC). We use network latencies between edge servers within 10–250 ms, a typical latency range between users and datacenters globally [21]. We instruct the schedulers (via the SLA) to find workers that satisfy 1 CPU, 100 MB memory, ≈ 20 ms latency (usual for immersive edge applications [46]), and 120 km operational distance. Since ROM only performs a best-fit match for computational requirements, its calculation time does not increase significantly while increasing the number of workers. LDP's calculation time grows increasingly with infrastructure size. However, LDP can effectively support latency-based constraints since it usually satisfies the RTT thresholds even in large edge infrastructures, which implies that the search space increases with the number of nodes. We leave a detailed comparison of different scheduling algorithms to future work as their performance is not the focal point of this paper.



(a) Tracking (b) Detection (c) FPS

Figure 9: Video analytics application performance.

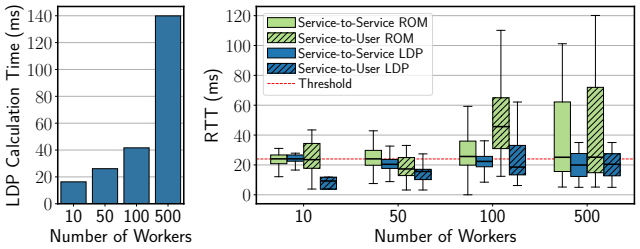


Figure 10: Performance of ROM and LDP schedulers.

5.5 Control Communication Overhead

Figure 11 shows a comparison between the number of exchanged control messages of `Oakestra` and K3s. First, we compare the K3s master and `Oakestra`'s cluster orchestrator for an apples-to-apples comparison. Then, we compare the control messages exchanged at the worker level. We record the network messages using the `strace`. Since K3s is a derivative of Kubernetes, they both use similar control communication. Both `Oakestra` and K3s workers send periodic updates to the master and receive control commands in return. However, the number of control messages ingress/egress on K3s far exceeds `Oakestra` (≈ 2×). These results help explain the observations presented in fig. 7. Moreover, a larger amount of control traffic also influences the time needed to synchronize the infrastructure and perform a deployment. Specifically, the master of "kube" based frameworks sends frequent messages to the workers, including specification of the pods to be attached to the workers, liveness checks, etc., requiring

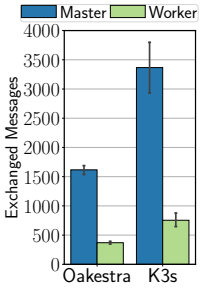


Figure 11: Control message overhead.

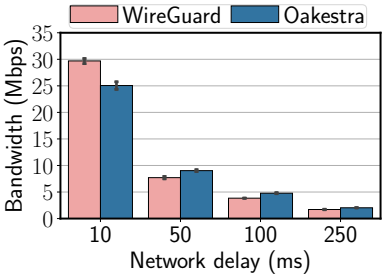


Figure 12: Oakestra vs. WireGuard tunneling overhead.

information/acknowledgment in return. The control communication of Oakestra, on the other hand, is simplified to send periodical aggregated service information from the worker over MQTT messages and keep a minimal footprint. It must be considered that Oakestra still does not provide the guarantees of a production-ready platform. Increasing the monitoring capacity and the update frequency will result in an increase of control traffic. We, therefore, plan on improving the control message communication channel in the future by dynamically tuning the update frequency depending on network conditions.

5.6 Video Analytics Application

We deploy the four microservices composing the video analytics pipeline described earlier on four *S* VMs in the HPC testbed. The provisioned resources do not support GPU acceleration and are single-core machines; therefore, the resulting FPS output is expected to be low. This setup is supposed to stress the platforms into executing this relatively heavy workload on extremely constrained resources. Both K8s and MicroK8s could not support the application since their orchestration components consumed most of the hardware capacity. As a result, we compare application performance over Oakestra, K3s, and without orchestration (native), with native acting as baseline (see fig. 9). Oakestra and K3s exhibit similar performance for object tracking, taking $\approx$ 300-400 ms. However, due to its minimal footprint Oakestra significantly outperforms K3s for supporting the more resource-demanding object detection service, achieving results closer to the baseline. Overall, application performance over Oakestra exceeded K3s by almost 10%. We omit our HET testbed results since they performed similarly to HPC.

6 Conclusion

We presented Oakestra, a flexible hierarchical orchestration framework designed for heterogeneous and constrained edge computing infrastructures. With its logical management hierarchy, Oakestra can sustain a high degree of context separation at scale. The delegated service scheduling of Oakestra reduces the task placement complexity and allows the framework to dynamically adjust to infrastructure changes irrespective of the scale. Furthermore, the proposed networking component enables developers to seamlessly and dynamically adjust the balancing policy with minimal overhead while natively being able to cross different networking boundaries. The lightweight implementation of Oakestra allows it to easily manage constrained resources likely to operate as "edge servers". In such contexts Oakestra superseded the performance of popular production frameworks (Kubernetes and its derivatives), achieving $\approx 10\times$ resource usage reduction and 10% application performance improvement.

We plan to add several feature extensions to Oakestra. For example, we aim to dynamically assign the cluster orchestrator role upon failovers via distributed leader election. We also intend to extend the scheduling and networking capabilities of Oakestra with recent research solutions for the edge, e.g. deadline-aware scheduling, multi-level tunneling, etc. To provide better QoS guarantees, we also aim to support and compare more recent application runtimes such as unikernels, demikernels, or Akka. Finally, we also seek to explore the possibility to incorporate Kubernetes deployments as Oakestra's clusters to achieve integration for existing cloud deployments.

Acknowledgments

We would like to thank the anonymous ATC reviewers and the shepherd for their helpful comments and insights during the review process of this paper. We would also like to thank the ATC Artifact Evaluation Committee for their meticulous examination and efforts to reproduce our results. We are also thankful to Hasso Plattner Institute (HPI) Germany for providing us access to their infrastructure which was instrumental for our experiments. This work was partly supported by the Federal Ministry of Education and Research of Germany (BMBF) project 6G-Life (16KISK002) and EU Health and Digital Executive Agency (HADEA) program under Grant Agreement No 101092950 (EDGELESS project).

References

[1] Celery - distributed task queue. https://docs.celeryq.dev/en/stable/.

[2] Containerd - an industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io.

[3] Istio - simplify observability, traffic management, security, and policy with the leading service mesh. https://istio.io.

[4] Jwt tokens. https://jwt.io.

[5] Mongodb documentation. https://www.mongodb.com/docs/.

[6] Mosquitto mqtt broker. https://mosquitto.org.

[7] Openapi initiative. https://www.openapis.org.

[8] Qemu - a generic and open source machine emulator and virtualizer. https://www.qemu.org.

[9] Submariner - a tool built to connect overlay networks of different kubernetes clusters. https://github.com/submariner-io/submariner.

[10] Kubernetes components, Mar 2021.

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

[12] Maha Aljarah, Mohammad Shurman, and Sharhabeel H Alnabelsi. Cooperative hierarchical based edge-computing approach for resources allocation of distributed mobile and iot applications. *International Journal of Electrical and Computer Engineering (IJECE)*, 10(1):296–307, 2020.

[13] Simon Bäurle and Nitinder Mohan. Comb: A flexible, application-oriented benchmark for edge computing. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '22, page 19–24, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, Renton, WA, April 2022. USENIX Association.

[15] Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *ZEUS*, pages 65–73, 2021.

[16] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. How to place your apps in the fog: State of the art and open challenges. *Software: Practice and Experience*, 50(5):719–740, 2020.

[17] Tatjana Chavdarova, Pierre Baque, Stephane Bouquet, Andrii Maksai, Cijo Jose, Timur Bagautdinov, Louis Lettry, Pascal Fua, Luc Van Gool, and Francois Fleuret. WILDTRACK: A Multi-camera HD Dataset for Dense Unscripted Pedestrian Detection. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5030–5039, Salt Lake City, UT, June 2018. IEEE.

[18] Claudio Cicconetti, Marco Conti, and Andrea Passarella. A decentralized framework for serverless edge computing in the internet of things. *IEEE Transactions on Network and Service Management*, 18(2):2166–2180, 2021.

[19] Cloud Native Computing Foundation (CNCF). Kubeedge. https://github.com/kubeedge/kubeedge, 2022.

[20] Google Coral. Coral edge. https://coral.ai/products, 2022.

[21] Lorenzo Corneo, Maximilian Eder, Nitinder Mohan, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, Per Gunningberg, Jussi Kangasharju, and Jörg Ott. Surrounded by the clouds: A comprehensive cloud reachability study. In *Proceedings of the Web Conference 2021*, WWW '21, page 295–304, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Lorenzo Corneo, Nitinder Mohan, Aleksandr Zavodovski, Walter Wong, Christian Rohner, Per Gunningberg, and Jussi Kangasharju. (how much) can edge computing change network latency? In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2021.

[23] Michael Cote. Why large organizations trust kubernetes. https://tanzu.vmware.com/content/blog/why-large-organizations-trust-kubernetes, 2020.

[24] Vittorio Cozzolino, Leonardo Tonetto, Nitinder Mohan, Aaron Yi Ding, and Jorg Ott. Nimbus: Towards latency-energy efficient task offloading for ar services. *IEEE Transactions on Cloud Computing*, pages 1–1, 2022.

[25] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, aug 2004.

[26] The Khang Dang, Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Jörg Ott, and Jussi Kangasharju. Cloudy with a chance of short rtts: Analyzing cloud connectivity in the internet. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 62–79, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Jason A. Donenfeld. WireGuard. https://www.wireguard.com/, 2022.

[28] Janick Edinger, Martin Breitbach, Niklas Gabrisch, Dominik Schäfer, Christian Becker, and Amr Rizk. Decentralized low-latency task scheduling for ad-hoc computing. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 776–785, 2021.

[29] Cloud Native Computing Foundation. Lightweight kubernetes | k3s. https://k3s.io, 2022.

[30] Eclipse Foundation. Eclipse fog05. https://fog05.io/, 2022.

[31] Eclipse Foundation. Eclipse iofog. https://iofog.org/, 2022.

[32] Linux Foundation. Scheduling and eviction | kubernetes. https://kubernetes.io/docs/concepts/scheduling-eviction/, 2022.

[33] Raspberry Pi Foundation. Raspberry pi 4. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/, 2022.

[34] The Linux Foundation. Kubernetes, 2022.

[35] The Linux Foundation. Kubernetes cluster federation | kubefed. https://github.com/kubernetes-sigs/kubefed, 2022.

[36] Intel. Intel nuc. https://www.intel.com/content/www/us/en/products/details/nuc/boards.html, 2022.

[37] Andrew Jeffery, Heidi Howard, and Richard Mortier. Rearchitecting kubernetes for the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, page 7–12, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Youngjin Kim, Chiwon Song, Hyuck Han, Hyungsoo Jung, and Sooyong Kang. Collaborative task scheduling for iot-assisted edge computing. *IEEE Access*, 8:216593–216606, 2020.

[39] Daniel King and Adrian Farrel. A Survey of Semantic Internet Routing Techniques. Internet-Draft draft-king-irtf-semantic-routing-survey-03, Internet Engineering Task Force, November 2021.

[40] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Juan Liu, Yuyi Mao, Jun Zhang, and Khaled B. Letaief. Delay-optimal computation task scheduling for mobile-edge computing systems. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 1451–1455, 2016.

[42] Qiang Liu and Tao Han. Virtualedge: Multi-domain resource orchestration and virtualization in cellular edge computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1051–1060, 2019.

[43] Canonical Ltd. Lightweight kubernetes | microk8s. https://microk8s.io, 2022.

[44] Anil Madhavapeddy and David J. Scott. Unikernels: The rise of the virtual library operating system. *Commun. ACM*, 2014.

[45] Karim Manaouil and Adrien Lebre. *Kubernetes and the Edge?* PhD thesis, Inria Rennes-Bretagne Atlantique, 2020.

[46] Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. Pruning edge research with latency shears. HotNets '20, page 182–189, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Nitinder Mohan and Jussi Kangasharju. Edge-fog cloud: A distributed cloud for internet of things computations. In *2016 Cloudification of the Internet of Things (CIoT)*, pages 1–6, 2016.

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

[49] Shadi A. Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Comp. and Comm.*, 23(4):11–20, may 2020.

[50] Nvidia. Jetson agx xavier. https://www.nvidia.com/en-us/autonomous-machines/jetson-agx-xavier/, 2022.

[51] Oakestra. Oakestra - discord. https://discord.gg/7F8EhYCJDf, 2023.

[52] Oakestra. Oakestra - github. https://github.com/oakestra, 2023.

[53] Oakestra. Oakestra artifacts - experiments. https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts/tree/main/Experiments, 2023.

[54] Oakestra. Oakestra artifacts - main. https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts, 2023.

[55] Oakestra. Oakestra artifacts - network. https://github.com/oakestra/USENIX-ATC23-Oakestra-net-Artifacts, 2023.

[56] Oakestra. Oakestra wiki. https://www.oakestra.io/docs/, 2023.

[57] M. Krochmal S. Cheshire. Rfc 6762 - multicast dns. 2013.

[58] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 168–178, 2016.

[59] Farah Aït Salaht, Frédéric Desprez, and Adrien Lebre. An overview of service placement problem in fog and edge computing. *ACM Comput. Surv.*, 53(3), jun 2020.

[60] Enrique Saurez, Harshit Gupta, Alexandros Daglis, and Umakishore Ramachandran. Oneedge: An efficient control plane for geo-distributed infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 182–196, New York, NY, USA, 2021. Association for Computing Machinery.

[61] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[62] Tanya Shreedhar, Sanjit K. Kaul, and Roy D. Yates. An age control transport protocol for delivering fresh updates in the internet-of-things. In *2019 IEEE 20th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 1–7, 2019.

[63] Manoel C Silva Filho, Claudio C Monteiro, Pedro RM Inácio, and Mário M Freire. Approaches for optimizing virtual machine placement and migration in cloud environments: A survey. *Journal of Parallel and Distributed Computing*, 2018.

[64] SuperMicro. Outdoor edge systems. https://www.supermicro.com/en/products/outdoor-edge, 2022.

[65] Dianomic Systems. Foglamp – simplifying iiot data management from sensors to clouds. https://dianomic.com/platform/foglamp/, 2022.

[66] Li Tianze, Wu Muqing, Zhao Min, and Liao Wenxing. An overhead-optimizing task scheduling strategy for ad-hoc based mobile edge computing. *IEEE Access*, 5:5609–5622, 2017.

[67] 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*, SEC '17, New York, NY, USA, 2017. ACM.

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

[69] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1270–1278, 2019.

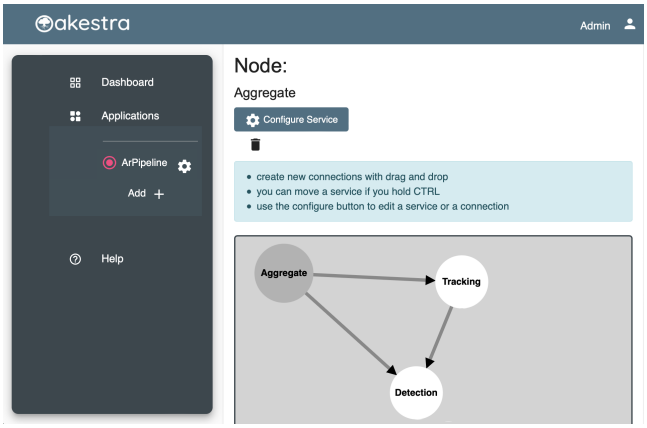
A `Oakestra` Frontend

Oakestra also provides a front-end application that can be used by the developers to create applications and describe the services composing them graphically. A developer can generate the SLA of each microservice through a form. Then, the services connections can be specified using the connection graph (fig. 13a). This graph allows developers to interconnect the services and specify their latency requirements. Once the connection graph is complete, the developer can ask the infrastructure to schedule the desired application. Via the interface, the developer can check each service instance's status and position (fig. 13b); they can also scale the services up, down, or terminate them.

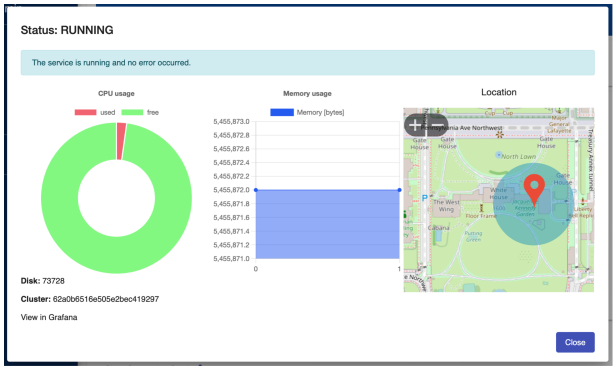
B Artifact Appendix

Abstract

`Oakestra` is an open-source project with all components publicly available on GitHub at https://github.com/oakestra. To ensure the reproducibility of the experiments



(a) Service connection graph



(b) Deployment monitoring interface

Figure 13: `Oakestra` web-based front-end application

conducted within this paper, we fork our project repositories at [54] and [55]. Additionally, we also provide a comprehensive `README`, which includes a `get-started` guide, that can be used for recreating our setup, experiments and familiarizing with the `Oakestra` platform [53].

Scope

The proposed artifacts represents a snapshot of the project that aligns with the paper. The proposed artifacts enable replicating the performance results of `Oakestra` as shown in section §5. However, if the reader is planning to use the latest version of the platform and utilize `Oakestra`'s latest functionalities, we recommend exploring the official website and repository instead.

Contents

Figure 14 shows how the components introduced in §3 can be related to the github repositories. Specifically, the source code, the release binaries and container images are split into the repositories as follows:

- `oakestra/oakestra` [54]: This repository contains the Root & Cluster orchestrators folders, as well as the Node Engine source code for the worker node. Inside the root orchestrator folder, the folders `system-manager-python/` and `cloud_scheduler/` contain the *System Manager* and the *Cloud Scheduler* source code, respectively. Similarly, the *Cluster Orchestrator* folder includes the source of the `cluster-manager/` and the `cluster-scheduler/`. Finally, `go-node-engine/` contains the implementation of the *Node Engine*.

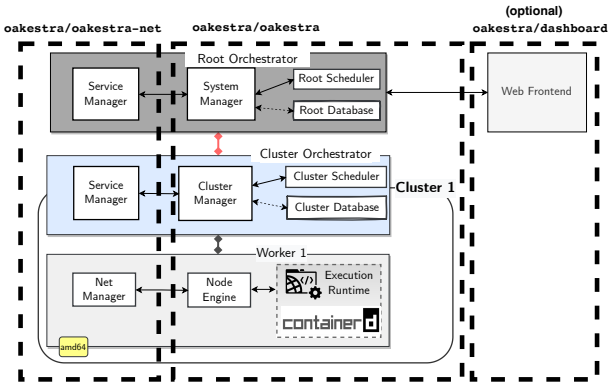


Figure 14: Summary of how the components are split across the repositories

- `oakestra/oakestra-net` [55]: This repository contains the Root, Cluster, and Worker *network* components. Note that the networking stack is not mandatory in `Oakestra`. Without `oakestra-net`, the developer can deploy applications on the infrastructure, but the applications will not be able able to carry out network-related tasks. Since our experiments utilize network-capable applications, our experiment's `README` provides detailed instructions regarding the installation of these components.

- `experiments` [53]: To replicate the experiments, we provide an `Experiments/` folder that includes the setup instructions to create your first `Oakestra` infrastructure and a set of scripts to automate the results collection procedure once the infrastructure is set up as shown in Table 1.

Requirements

We recommend the following minimum requirements for each `Oakestra` component.

1. *Root Orchestrator*: 2GB of RAM, 2 Core CPU, ARM64 or AMD64 architecture, 10GB disk, docker compose installed. Tested OS: Ubuntu 20.20, Windows 10, MacOS Monterey.

Folder	Description
`Test 1`	Deployment overhead calculation (figs. 5 and 7)
`Test 2`	Network overhead measurements (fig. 8)
`Test 3`	Bandwidth measurements (fig. 12)
`Test 4`	Control message measurements (fig. 11)
`Test 5`	Scalability stress test experiments (fig. 6)
`Test 6`	AR pipeline experiments (fig. 9)

Table 1: Experiments test folders

2. *Cluster Orchestrator*: 2GB of RAM, 2 Core CPU, ARM64 or AMD64 architecture, 5GB disk. Tested OS: Ubuntu 20.20, Windows 10, MacOS Monterey.

3. *Worker Node*: Linux Machine, 1 Core CPU, ARM64 or AMD64 architecture, 2GB disk, `iptables` utility.

Beyond the Paper

The AE repository only contains the performance evaluation of `Oakestra`. In addition to this appendix and the repo, the `Oakestra` project provides extensive documentation [56] on how to use `Oakestra` in a variety of infrastructure configurations and applications. In addition, interested researchers are welcome to join the community via GitHub [52] or Discord [51].

Explore Data Placement Algorithm for Balanced Recovery Load Distribution

Yingdi Shan[1,2], Kang Chen[2] and Yongwei Wu[2]

[1]Zhongguancun Laboratory
[2]Tsinghua University

Abstract

In distributed storage systems, the ability to recover from failures is critical for ensuring reliability. To improve recovery speed, these systems often distribute the recovery task across multiple disks and recover data units in parallel. However, the use of fine-grained data units for better load balancing can increase the risk of data loss.

This paper systematically analyzes the recovery load distribution problem and proposes a new data placement algorithm that can achieve load balancing without employing fine-grained data units. The problem of finding an optimal data placement for recovery load balancing is formally defined and shown to be NP-hard. A greedy data placement algorithm is presented, and experimental results demonstrate its superior performance compared to conventional techniques, with up to 2.4 times faster recovery. Furthermore, the algorithm supports low-overhead system expansion.

1 Introduction

In distributed storage systems, data is divided into smaller units called data units, which are grouped together in a placement group for reliability. For example, Google File System (GFS) [6] uses 64MB chunks as data units and replicates each chunk three times to form a placement group. The technique of erasure coding [7, 14, 18] can be utilized to calculate the parity of data units that have been grouped together to form a placement group, thereby providing another method for data reliability. If a single node fails in the storage system, the lost data units on that node must be repaired and distributed to other nodes, a process known as recovery. In this context, the term node is used to refer to an entity within the distributed system, which can be either a whole server or a single disk. Since there are many data units on a single node and different data units on the same node may belong to different placement groups, these storage systems can perform recovery in parallel, improving recovery speed [12, 13, 21]. Parallelized recovery has the potential to increase recovery speed so that it is limited by the cluster's bandwidth rather than the bandwidth of an individual node, improving the overall reliability of the storage system.

However, parallelized recovery does not necessarily imply faster recovery. An imbalance in the distribution of recovery load among various nodes can lead to congestion in certain nodes and prolonged recovery times. Imbalanced recovery loads can also negatively impact the performance of certain nodes, as they may have to devote a significant amount of their bandwidth to recovery.

To address these issues, distributed storage systems often employ fine-grained data units to balance the recovery load [6, 20]. By distributing a sufficient number of data units across various nodes, the recovery load can be evenly distributed through randomization [13]. However, this approach to fine-grained recovery, while effective in load balancing, also increases the risk that any placement group in the cluster fails [1–4, 9, 22]. Furthermore, the utilization of fine-grained data units can result in an uptick in overhead for the management of metadata.

This paper presents a novel data placement algorithm designed to achieve a more balanced recovery performance without the need for fine-grained data units. Data placement, which refers to the mapping of data units to disks, is challenging to design as it impacts data distribution and system scaling. Therefore, an effective data placement algorithm that can balance the recovery load should not have any negative impact on these perspectives.

To create a data placement algorithm for balancing the recovery load, this paper begins by formally defining the optimal recovery load distribution problem as selecting a set of nodes to minimize the weight of the edge with the highest weight in the recovery load graph. We then prove that this problem is NP-hard by showing that it can be transformed into a maximum independent set problem in polynomial time.

The paper then proposes a data placement algorithm based on a greedy strategy that is able to compute a more balanced data placement for recovery load distribution. The algorithm also supports low-overhead system expansion. Experiments demonstrate that the use of this data placement algorithm can improve the storage system's recovery performance by 1.7-2.4 times compared to the original data placement algorithm.

The contributions of this paper include:

- We formally define the optimal recovery load distribution problem. Following this, the paper proves that the problem is NP-hard.
- We propose a greedy data placement algorithm for efficient recovery. It is experimentally demonstrated that the algorithm is able to provide a more balanced recovery load distribution. The algorithm can support low-overhead system expansion.

2 Problem Definition and Analysis

2.1 Repair Load Matrix

The utilization of both replication and erasure coding techniques serves as a means to guarantee reliability in data storage systems. Various erasure codes exist, each having distinct read patterns. For instance, LRC (Local Reconstruction Codes) [7] utilize a minimal number of nodes for repair, while regenerating codes [5, 18] necessitate the read of data from a larger number of nodes but only a fraction of the information from each node. To harmonize these techniques, the repair load matrix is employed as a means to describe the repair process for both replication and erasure coding.

The repair load matrix is a matrix that represents the cost required to repair a failed node. In scenarios where I/O is the system bottleneck, the cost typically refers to I/O expenses, and nodes are often interpreted as individual disks. Conversely, when network bandwidth is the system's bottleneck, the cost usually denotes network expenditures, with nodes commonly referring to whole servers.

Let us denote the cardinality of a placement group as n. Correspondingly, the repair load matrix is represented by an $n \times n$ matrix W. The element $W_{i,j}$ within this matrix signifies the cost incurred to retrieve data from the jth node in the placement group when the ith node within the same group encounters a failure. For example, the repair load matrix for a $(k = 3, r = 1)$ Reed-Solomon (RS) code [14] is given below, and it shows that when any node fails, data needs to be read from all other nodes, and the cost of reading data from any other node is 1.

$$W = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

In the context of an RS code where the redundancy parameter r is greater than 1, there exists a multitude of plausible options for the repair load matrix W. This is contingent upon which nodes are selected for data reading during the recovery process. Any such selection can appropriately serve as the repair load matrix for the system.

For a $(k = 4, r = 2, l = 2)$ LRC code, the repair load matrix can be written as

$$W = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

In this matrix, when repairing a data node or local parity node, the LRC needs to read data from $\frac{k}{l}$ nodes in the same group in order to reconstruct the original data. When a global parity node requires repair, the LRC must read all data nodes' contents.

2.2 Recovery Load Graph

The repair load matrix defines the repair process for a single placement group within a distributed storage system. In contrast, the recovery load graph (or recovery load matrix) characterizes the recovery load on each node in the event of a failure within the system. This is achieved by summing the repair load for each placement group. The recovery load graph is defined mathematically as follows:

Definition 1 (Recovery Load Graph)**.** *Let W be the repair load matrix defined above. Suppose there are N nodes in the storage system, and the storage system has S placement groups, denoted $P_1, P_2, \ldots, P_S$. Each placement group P_i is an array with length n, where each value, denoted by $P_{i,k}$, represents a node id. If $P_{t,k} = i$, then we define $Index(P_t, i) = k$. The recovery load graph corresponding to this storage system is $G = (V, E)$, where V is the set of points in the graph with cardinality N, representing the nodes in the system, and E is the set of edges in the graph. Let $E_{i,j}$ denote the edge weight between node i and node j, representing the load of reading data from node j when node i fails. Let $[i \in P]$ return 1 if $i \in P$ and 0 otherwise. The edge weight is calculated as follows:*

$$E_{i,j} = \sum_{t=1}^{S} [i \in P_t] \cdot [j \in P_t] \cdot W_{Index(P_t,i),Index(P_t,j)} \qquad (1)$$

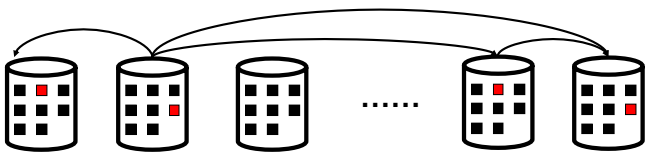


Figure 1: Recover Load Graph

2.3 Optimal Recovery Load Distribution

In this paper, we consider a system in which each placement group comprises an equal quantity of data. Placement groups that are not at capacity, as they consume a minimal amount of storage space, are disregarded. Upon the arrival of new data, the system must determine an appropriate placement group to accommodate it. The objective is to minimize the imbalanced recovery load upon the addition of a new placement group to the system. An alternative approach to this problem would be to determine the optimal recovery load distribution by considering all placement groups simultaneously, as opposed to incrementally identifying a single placement group. However, such an approach may significantly restrict the system's ability to add or remove a node, as it is hard to predict these events in advance.

An optimal distribution of recovery loads should strive to minimize the maximum recovery load after the integration of a new placement group. The definition of the optimal recovery load distribution problem is given below:

Definition 2 (Optimal Recovery Load Distribution Problem)**.** *Given a recovery load graph $G = (V, E)$, the optimal recovery load distribution problem is to construct an array P of length n such that all its elements belong to V, and the corresponding set of edges E', where $E'_{i,j} = E_{i,j} + [i \in P] \cdot [j \in P] \cdot W_{Index(P,i),Index(P,j)}$, such that the maximum value in the edge set E' is minimized. The optimal recovery load corresponding to graph G is:*

$$\min_{P} \max_{i,j \in V} E'_{i,j} \tag{2}$$

As shown in Figure 1, when node i and node j have a common placement group, the edge weight between them needs to be added to the repair cost corresponding to the repair load matrix. The optimal recovery load distribution problem then becomes a problem of selecting n nodes from the N nodes in graph G, such that the highest edge weight in the graph is minimized after the weights are added.

2.4 Complexity Analysis and Proof

In this section, we prove that the optimal recovery load distribution problem is NP-hard by reducing a known NP-complete problem, the maximum independent set problem [8], to it within polynomial complexity.

Definition 3 (Maximum Independent Set Problem)**.** *Given a graph $G = (V, E)$ and an integer n, let $N(i)$ represent the set of neighbor nodes of node i. The decision form of the maximum independent set problem is to determine whether there exists a set $P \subseteq V$ with cardinality not less than n such that $\forall i \in P$, $N(i) \cap P = \emptyset$.*

As shown in Figure 2, an independent set of a graph is a set of nodes such that no two nodes in the set have an edge between them. The red nodes in Figure 2 form an independent set of the graph because there are no edges between any two colored nodes. The maximum independent set problem is to find an independent set of cardinality not less than n in the graph.

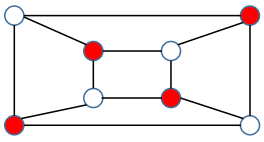


Figure 2: Maximum independent set problem

Lemma 1. *The maximum independent set problem can be reduced to the optimal recovery load distribution problem in polynomial time.*

Proof. We consider the following problem for an arbitrary graph $G = (V, E)$: Given the recovery load graph $G' = G = (V, E)$, where W is a repair load matrix with all entries equal to 1 except for the diagonal entries. The recovery load edge weight $E_{i,j} = 1$ when there is an edge in G from node i to node j. The goal is to find a placement group P that minimizes the maximum value in the updated edge set E' of graph G' after the integration of a new placement group.

We prove that the problem of finding the maximum independent set of graph G can be reduced to solving the optimal recovery load problem. Specifically, we show that the cardinality of the independent set of graph G is not less than n **if and only if** there exists a placement group P with cardinality n such that the optimal recovery load of graph G' is 1.

If there exists an independent set of cardinality not less than n in graph G, we can simply select n nodes from the independent set and place the new placement group on these n nodes in G'. Denote this set of n nodes as p. Since these n nodes are not adjacent to each other, $E_{i,j} = 0$ and $E'_{i,j} = 1$ for any node i and node j that belongs to p. Any other edges in E' remain the same as in E, which are no larger than 1. Therefore, the edge with the largest weight in E' will not exceed 1. As a result, the optimal recovery load corresponding to graph G' is 1.

Conversely, if the optimal recovery load corresponding to graph G' is 1, we assume that there is no independent set of cardinality not less than n in graph G. This means that we can arbitrarily select n nodes from the graph G, and at least two of these n nodes are adjacent. Then, if we select the same set from the graph G', the maximum edge weight of the edge $E_{i,j}$ in this set is 1. After the integration of the new placement, the edge weight of the edge $E'_{i,j}$ is at least 2 because W is a repair load matrix with all entries equal to 1 except for the diagonal entries, which suggests that the optimal recovery load for this graph is at least 2. This contradicts the premise, so it follows that there must exist an independent set of cardinality not less than n in the graph $G = (V, E)$.

Additionally, the reduction of the problem has a complexity of $O(|E|)$, as each edge of the graph G can be transformed into an edge in the recovery load graph by visiting it once. Thus, the problem of finding the maximum independent set can be efficiently reduced to the problem of determining the optimal recovery load distribution in polynomial time. □

Since the maximum independent set problem is an NP-complete problem [8], it follows from Lemma 1 that the optimal recovery load distribution problem is an NP-hard problem.

3 Algorithm Design

3.1 Data Placement Algorithm

This section presents a data placement algorithm based on a greedy strategy for the NP-hard problem of optimal recovery

load distribution. The algorithm aims to select nodes for a placement group of size n from a set of N nodes by choosing, at each step, the node whose sum of recovery costs to other nodes in the current group is smallest. While this method may not necessarily result in the optimal recovery load distribution, it can help to balance the recovery load.

The pseudocode for the algorithm is provided in Algorithm 1. This algorithm is responsible for mapping placement groups to nodes, and it is called to obtain the placement group P whenever a new placement group needs to be added to the storage system. The input to the algorithm is the current recovery load graph $G(V,E)$. The weight $WV(v)$ of a vertex v is defined as the sum of the weights of its adjacent edges, that is, $WV(v) = \sum_{e \in neighbor(v)} E(e)$. $D(v)$ is the number of data units on the node. Capital letters, such as V_0, represent sets, while lowercase letters, such as v_1, represent members of a set. For example, V_0 is a set of vertices and v_0 is a vertex within that set. The function *Pick* is used to randomly select a member from a set. The output of the algorithm is a placement group P.

Algorithm 1: Greedy Data Placement Algorithm

Input : $G(V,E)$
Output : P

1 $v_0 = Pick(\{v \mid WV(v) = min_{v' \in V}(WV(v'))\})$;
2 $P = \{v_0\}$;
3 **while** $|P| < n$ **do**
4 $\quad V_{candidates} = V - P$;
5 $\quad V_0 = V_{candidates} - \{v \mid violates\ criteria\}$;
6 $\quad V_1 = \{v \mid D(v) \leq (min_{v_0 \in V_0} D(v_0)) \cdot (1+\varepsilon) \wedge v \in V_0\}$;
7 $\quad V_2 = \{v \mid \sum_{v' \in P} E_{v,v'} = min_{v_1 \in V_1}(\sum_{v' \in P} E_{v_1,v'}) \wedge v \in V_1\}$;
8 $\quad v_{next} = Pick(V_2)$;
9 $\quad P = P \cup \{v_{next}\}$;
10 Update weight in G;

The proposed data placement algorithm based on the greedy strategy must exclude certain nodes from being placed in the same placement group. For example, nodes in the same placement group should not be on the same server or cabinet in order to reduce the risk of data loss due to systemic failures. Line 5 of the algorithm demonstrates how the candidate set V_0 is constructed based on this rule.

Heuristic rules are used to find the most suitable next node for the current placement group. This algorithm first selects the node with the smallest edge weight sum as the initial node. This process is shown by lines 1 to 2 of the algorithm. In order to achieve a uniform distribution of data, in line 6, V_1 is defined as a vertex whose number of data units does not exceed $(1+\varepsilon)$ times the minimum number of data units in the current storage system. Since the data is difficult to achieve absolute uniform distribution, this algorithm uses an adjustable parameter ε to limit the uniformity of the data placement.

To distribute the recovery load more evenly among nodes, V_2 is defined as the set of nodes with the smallest sum of recovery costs to other nodes in the current replacement group in the event of a failure, as shown in line 7 of the algorithm. Finally, a vertex v_{next} is chosen from V_2 as the next candidate for the placement group and added to the set P. This process continues until n members have been added to the set P, after which the corresponding weights in G can be updated.

When a node represents a server instead of a disk, data units for each node in the group require assignment to a specific disk. This assignment can be efficiently achieved through a round-robin strategy.

3.2 Target Node Selection

When a single node fails and requires recovery, the data stored on it must be redistributed to other nodes in order to maintain the reliability of the system. During this process, it is crucial to select nodes to receive the data units from the failed node in a manner that ensures even distribution of recovered data and maintains balance in the recovery load. The algorithm in Algorithm 2 can be used to select the location of data units to be redistributed to other nodes during the recovery process. This algorithm first excludes nodes that do not meet certain criteria as in Algorithm 1. Next, the algorithm looks for nodes with used storage space within a certain range in order to achieve a balanced distribution of data, as described in line 2. The algorithm then selects nodes that will help maintain recovery load balance after the faulty node is removed, as shown in line 3. Finally, the algorithm aims to evenly distribute the load during recovery; therefore, if the previous conditions are met, it will select the node with the least connection weight to the failed node, as shown in line 4.

Algorithm 2: Target Node Selection for Recovey

Input : $G(V,E)$, placement group P, failure node F
Output : v

1 $V_0 = V - \{v \mid violates\ criteria\}$;
2 $V_1 = \{v \mid D(v) \leq (min_{v_0 \in V_0} D(v_0)) \cdot (1+\varepsilon) \wedge v \in V_0\}$;
3 $V_2 = \{v \mid \sum_{v' \in P} E_{v,v'} = min_{v_1 \in V_1}(\sum_{v' \in P} E_{v_1,v'}) \wedge v \in V_1\}$;
4 $V_3 = \{v \mid E_{v,F} = min_{v_2 \in V_2} E_{v_2,F} \wedge v \in V_2\}$;
5 $v = Pick(V_3)$;
6 Update weight in G;

Since each node may store multiple data units, Algorithm 2 must be run for each placement group to determine the target nodes.

3.3 System Expansion

When adding new nodes to the system, the graph G must be updated to include the nodes corresponding to these new devices. The system can then continue to call the algorithm

in Algorithms 1 to automatically add new placement groups containing the new nodes to the storage system, ensuring that the recovery load remains as small as possible without requiring any data migration.

However, when adding multiple nodes at once, using the algorithm in Algorithms 1 directly can result in all the new placement groups being placed on the same batch of newly added nodes. This can lead to an imbalanced recovery load, with a concentration of recovery load and writing load on the newly added devices, even as more placement groups are added. To address this issue, the algorithm should control the rate at which data is placed on the new nodes during system expansion.

To do this, the new nodes can be placed in a separate collection. Each time the system is expanded, the user can specify a parameter c to control the placement rate of data on the new devices. When calculating data placement, at most c nodes will be selected from this collection, with the remaining $n-c$ devices being selected from other devices according to the algorithm in Algorithm 1. Once the number of data units on a node within the collection matches the number of data units on the node with the least amount of data outside of this collection, the node can be removed from the collection.

4 Evaluation

4.1 Evaluation Setup

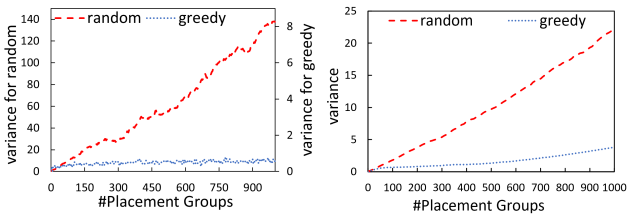
In the context of our evaluation, we designate each individual disk as a node within the system. We carry out both micro-benchmark experiments and overall performance testing to assess performance. The micro-benchmark experiments, performed outside of an actual storage system, specifically evaluate the variability in recovery load distribution and data distribution. Instead of executing actual recovery, micro-benchmark experiments provide an analysis of the recovery load distribution, which are complementary to our overall performance evaluation.

For the overall performance test, we employ a cluster of 16 servers, with each server boasting dual Intel Xeon E5 2643 v4 CPUs, 128GB of 2133 MHz DDR4 memory, a 512GB SATA3 SSD, and six 8TB 7200rpm SAS HDDs. The six HDDs are independent and not grouped by a RAID. All servers run on the CentOS 7.8 operating system. The servers are networked via a 56Gbps Infiniband connection, with an MTU size of 65520, enabling us to better utilize the high-bandwidth network in our evaluation. We set the value of ε to 0.02 for all experiments to maintain a consistent testing environment.

4.2 Micro Benchmark

We conduct micro-benchmark experiments to see how our algorithms can help to improve recovery load balance. In all experiments, N is set to 100, and RS(10,4) code is employed for each placement group.

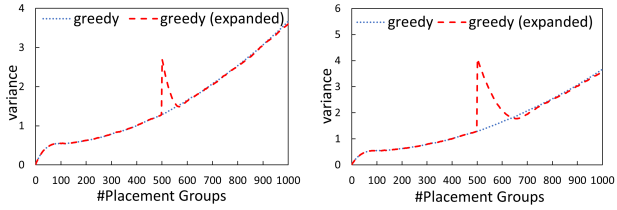
Data Distribution. Figure 3a measures the uniformity of data distribution for different data placement algorithms by calculating the variance of the number of data units. It can be seen that the greedy data placement algorithm is far more uniform than the random data placement algorithm.



(a) Data Distribution (b) Recovery Load Distribution

Figure 3: The variance of data distribution and recovery load distribution.

Recovery Load Distribution Figure 3b presents the recovery load distribution of the system when utilizing different data placement algorithms. The data depicted in the figure demonstrates that the recovery load is more evenly distributed when using a greedy data placement algorithm as compared to random data placement.



(a) Add a disk (b) Add two disks

Figure 4: The variance of the recovery load after system expansion.

Recovery Load Distribution for System Expansion We evaluate the effects of system expansion on recovery load distribution. In this experiment, the variable c is set to 1. The results are illustrated in Figure 4a and Figure 4b. Specifically, Figure 4a represents the scenario where an additional disk is added to the system when there are 500 placement groups stored, while Figure 4b illustrates the situation where two additional disks are added simultaneously.

The data illustrated in Figure 4a demonstrates that while the greedy algorithm may initially create a temporary imbalance in recovery load after system expansion, the recovery load returns to a normal state as more placement groups are integrated. This phenomenon arises due to the recovery load on newly introduced disks being zero initially, resulting in a sharp uptick in the variability of the recovery load distribution. However, as these new disks become populated with data, the recovery load re-establishes its balance.

Conversely, Figure 4b illustrates that simultaneous addition of multiple disks leads to a more pronounced imbalance in the

Table 1: The average recovery time of different data placement algorithms.

Codes	Random	Greedy	Improvement
RS code	554s	273s	2.1x
LRC	460s	192s	2.4x
Clay code	240s	141s	1.7x

recovery load of the storage system and that it takes longer for the system to return to normal.

4.3 Overall Performance

This experiment measures the recovery performance using different data placement algorithms and different erasure codes, including RS code, LRC, and Clay Code [18]. To achieve this, 96 hard disks were distributed across 16 machines, with each data unit set at a size of 10GB and a total of 175 placement groups in the system. The algorithms are integrated into the RCStor storage system [16], as it provides high recovery performance and various erasure codes. Prior to measuring recovery, data of approximately 256GB was placed on each disk, resulting in a total data size of 24.4TB. The recovery process was initiated manually by shutting down a disk, and the time required for recovery completion was recorded. To gauge the maximum recovery bandwidth, we initiated the recovery of all failed placement groups simultaneously. It should be noted that during recovery, the data from the failed disks was reconstructed on other functioning disks. We ensured there was no bandwidth cap for the recovery process. Additionally, the recovery was carried out at a time when the system was not in use, to avoid any operational interruptions. We conducted 10 trials to ascertain the average recovery time. The experimental results are presented in Table 1.

The data in Table 1 illustrates that the use of a greedy data placement algorithm can significantly enhance the recovery performance for various erasure codes. Specifically, the recovery performance is found to be 1.7-2.4 times greater than that of the random data distribution algorithm.

We have also evaluated the influence of randomness on recovery time. Our findings suggest that when utilizing the greedy data placement algorithm, the impact of randomness fluctuates within a range of ±10%. However, with the random data placement algorithm, this variability increases to ±20%.

5 Related Work

Copyset [4] proposes to reduce the probability of data loss by reducing the number of distinct placement groups, which are referred to as copysets. In contrast, the focus of our paper is to ensure recovery load balance given a predetermined small number of placement groups. This number is approximately the same as the number of copysets, contingent upon parameter configurations. In essence, we tackle the issue of data loss by balancing the recovery load given a fixed number of placement groups. This aspect marks a departure from the Copyset paper, which does not focus on recovery load balancing.

PDL [23] is a data placement algorithm that reduces imbalances in inter-cabinet network communication. SelectiveEC [24] maintains load balance during recovery by dynamically selecting nodes for reading and writing through bipartite graph matching. However, they only reduce recovery load imbalance for the network and cannot guarantee the balance of load when accessing disks.

RAID-based data placement algorithms [11, 15, 17, 19, 26] are designed for use with disk arrays, which are not appropriate for use in distributed storage systems. RAID+ [25] and D_3 [10] use orthogonal Latin squares to distribute data and ensure load balance for disk array recovery, but they lack scalability and can only be applied to arrays with tens or hundreds of disks. They also do not support dynamic system expansion.

Overall, existing data placement algorithms struggle to simultaneously provide load balancing for recovery, scalability and low migration overhead.

6 Discussion

In a storage system capable of tolerating multiple failures, resulting in numerous potential repair load matrices, a practical advantage may arise from dynamically selecting recovery sources, taking into account observed loads and stragglers. This dynamic selection aligns with our algorithm and could involve dynamically choosing k nodes from each placement group for data recovery, excluding stragglers, to minimize the maximal recovery load. The future challenge lies in developing an efficient algorithm for this process or devising other methods to account for the impacts of such dynamism.

7 Conclusion

This paper proposes a data placement algorithm based on a greedy strategy that can provide a more balanced recovery load distribution. Experiments show that using the data placement algorithm can improve the recovery performance of the storage system to 1.7-2.4 times compared to the random data placement algorithm.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Nathan Bronson, for their valuable comments and helpful suggestions. The authors from Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2022YFB4502004), Natural Science Foundation of China (62141216, 61877035) and Tsinghua University Initiative Scientific Research Program.

References

[1] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization for self-optimizing storage systems. In *7th USENIX Conference on File and Storage Technologies (FAST '09)*, pages 183–196, 2009.

[2] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures (SPAA '00)*, pages 119–128, 2000.

[3] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, pages 45–58, 2006.

[4] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC '13)*, pages 37–48, 2013.

[5] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, pages 29–43, 2003.

[7] 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 *2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 15–26, 2012.

[8] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[9] Jack YB Lee and John CS Lui. Automatic recovery from disk failure in continuous-media servers. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13(5):499–515, 2002.

[10] Zhipeng Li, Min Lv, Yinlong Xu, Yongkun Li, and Liangliang Xu. D3: Deterministic data distribution for efficient data reconstruction in erasure-coded distributed storage systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS '19)*, pages 545–556. IEEE, 2019.

[11] Alberto Miranda and Toni Cortes. Raid: Online raid upgrades using dynamic hot data reorganization. In *12th USENIX Conference on File and Storage Technologies (FAST '14)*, pages 133–146, 2014.

[12] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 1–15, 2012.

[13] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[14] 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.

[15] Beomjoo Seo and Roger Zimmermann. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Transactions on Storage (TOS)*, 1(3):316–345, 2005.

[16] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. Geometric partitioning: Explore the boundary of optimal erasure code repair. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, page 457–471, 2021.

[17] Lei Tian, Dan Feng, Hong Jiang, Ke Zhou, Lingfang Zeng, Jianxi Chen, Zhikun Wang, and Zhenlei Song. PRO: A popularity-based multi-threaded reconstruction optimization for RAID-Structured storage systems. In *5th USENIX Conference on File and Storage Technologies (FAST '07)*, pages 301–314, 2007.

[18] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. Clay codes: moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies (FAST '18)*, pages 139–154, 2018.

[19] Jiguang Wan, Jibin Wang, Changsheng Xie, and Qing Yang. S^2raid: Parallel raid architecture for fast data recovery. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(6):1638–1647, 2013.

[20] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. MAPX: Controlled data migration in the expansion of decentralized Object-Based storage systems. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*, pages 1–11, 2020.

[21] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06)*, pages 307–320, 2006.

[22] Qin Xin, Ethan L Miller, and SJ Thomas JE Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *13th IEEE International Symposium on High performance Distributed Computing (HPDC '04)*, pages 172–181, 2004.

[23] Liangliang Xu, Min Lv, Zhipeng Li, Cheng Li, and Yinlong Xu. PDL: A data layout towards fast failure recovery for erasure-coded distributed storage systems. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 736–745, 2020.

[24] Liangliang Xu, Min Lyu, Qiliang Li, Lingjiang Xie, and Yinlong Xu. SelectiveEC: Selective reconstruction in erasure-coded storage systems. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '20)*, 2020.

[25] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. RAID+: Deterministic and balanced data distribution for large disk enclosures. In *16th USENIX Conference on File and Storage Technologies (FAST '18)*, pages 279–294, 2018.

[26] Weimin Zheng and Guangyan Zhang. FastScale: Accelerate RAID scaling by minimizing data migration. In *9th USENIX Conference on File and Storage Technologies (FAST '11)*, pages 149–161, 2011.

LUCI: Loader-based Dynamic Software Updates for Off-the-shelf Shared Objects

Bernhard Heinloth, Peter Wägemann, and Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

Shared libraries indisputably facilitate software development but also significantly increase the attack surface, and when using multiple libraries, frequent patches for vulnerabilities are to be expected. However, such a bugfix commonly requires restarting all services depending on the compromised library, which causes downtimes and unavailability of services. This can be prevented by dynamic software updating, but existing approaches are often costly and incur additional maintenance due to necessary source or infrastructure modifications.

With LUCI, we present a lightweight linker/loader technique to unobtrusively and automatically update shared libraries during runtime by exploiting the indirection mechanisms of position-independent code, hence avoiding severe runtime overhead. LUCI further adds no additional requirements, such as adjusting the source or interfering with the build chain, as it fully adapts to today's build and package-update mechanisms of common Linux distributions. We demonstrate our approach on popular libraries (like *Expat* and *libxcrypt*) using off-the-shelf (i.e., unmodified) binaries from Debian and Ubuntu packages, being able to update the majority of releases without the necessity of a process restart.

1 Introduction

Third-party libraries are, without doubt, an important part of software development, allowing a programmer to use functionality beyond their domain (or at least to save some time). To prevent outdated source-code copies [49], it is common to dynamically link against libraries distributed in binary format [50] – on system-level so-called shared objects, enabling independent updates. A popular example is *OpenSSL*, providing cryptographic capabilities to numerous projects. However, its infamous *Heartbleed*-vulnerability demonstrated the downside: All of a sudden, millions of systems (yet alone 24 – 55% of all HTTPS-enabled servers [13]) ran the risk of leaking highly sensitive information when the buffer over-read bug was discovered. Even though distributors quickly published fixed versions of the library, all applications directly or indirectly depending on this library had to be restarted, not only requiring immediate manual action but also causing costly downtimes of services [12, 22].

Generally, having a new library version to be responsible for the need for a service restart is all but seldom: Default server software in many popular Linux distributions like Debian is dynamically linked and depends on several shared libraries, which are usually the main reason for a service being exposed to known vulnerabilities:

For example, the currently most widely used web server [23], *nginx*, had 11 (out of 34 CVEs[1]) vulnerabilities with high severity[2] since 2010, while its basic (static) dependencies *glibc*, *OpenSSL*, *PCRE*, *libxcrypt*, and *zlib* had at the same time 70 such critical vulnerabilities (of 335 CVEs total).

It is worth noting that these numbers do not take any modules into account: the default configuration of *nginx* includes six core modules, which depend on a total of over 30 external shared libraries themselves – and have at least 98 additional critical vulnerabilities (of 474 CVEs) in the observed period.

A similar picture emerges for the second place: 14 critical vulnerabilities were found in *Apache HTTP Server* including its core modules. In contrast, its required shared libraries in basic configuration (without modules) had 81 such issues due to an additional dependency on *Expat* compared to *nginx*.

Especially for stateful software systems (e.g., database management systems) or systems with active client connections, restarting the service in case of a vulnerability is undesired [36, 37]. To address the challenges of avoiding unwanted downtimes and expensive startup costs, dynamic software update (DSU) mechanisms have been developed over the last decades [8, 11, 15, 28, 29, 42].

Although many interesting DSU techniques are available, and even Linux introduced kernel live-patching in version 4.0 more than seven years ago [20], the requirement for restarting services utilizing updated libraries has not changed since

[1] Common Vulnerabilities and Exposures: www.cve.org.
[2] CVSS (Common Vulnerability Scoring System) score with 7.0 or higher.

then. Unfortunately, common user-space live-patching has not become a reality yet.

This raises the question regarding the reasons for this shortcoming. Most approaches for DSU require modifications in the source [15, 17, 28, 29] or build system [11, 18, 42, 47], programmer-guided patch creation [34, 43, 45], and/or a virtual machine [5, 7, 35] for execution. While the modification of a single project can be rather easy, it would have to be applied to almost every software package in a system for practical usage of dynamic updates – and yet the vast amount of different software prevents any broad application.

That is why we believe that system-wide automatic live updates in user space will only succeed if the requirements are limited to the bare minimum: We argue that *unmodified (i.e., off-the-shelf) binary files* already built and deployed by distributors have to be sufficient input for base and updated versions.

In addition, imposed runtime overhead and stability concerns due to the complexity of existing approaches further hinder any attempts to establish general live-patching in user space. Nevertheless, these penalties are inevitable fallout of the effort to provide a general approach allowing to transform almost any program state – which is usually not required for security fixes: Since the vast majority of system-level software is still written in programming languages not ensuring memory safety (like C and C++) [6], patches for critical vulnerabilities commonly introduce small local code changes like bounds checking (e.g., the fix for the mentioned *Heartbleed* bug). Such patches usually do not alter any function output for valid inputs [24].

The situation for logic errors is similar: Most common bugfix patterns affect only the function scope [32] without side effects beyond its borders. Especially system-level shared objects have to maintain the *application binary interface* (ABI) to which other software binaries are dynamically linked, making structural changes rather seldom. New feature improvements altering the library's *application programming interface* (API) can usually only be used in depending software after modifying its source code as well. Since adjusting projects to the libraries' updated semantics may take time [21], distributors tend to backport bugfix patches while retaining API & ABI compatibility [39].

These insights help us to define the necessary scope required for dynamic software updates based on binary files under practical conditions: First, we focus on supporting the mentioned changes required for error correction instead of enabling updates to introduce arbitrary modifications. Second, we argue that a practical DSU solution has to update off-the-shelf binaries without requiring access to or modifications of the source code or build process.

With LUCI, we present a DSU approach for unmodified shared libraries utilizing features already enabled by default in today's build chains without inducing runtime overhead. In this sense, the paper makes the following contributions:

- A lightweight loader-based DSU mechanism targeting security fixes that is based on relinking dynamic ELF binaries by leveraging its metadata
- Design and implementation of an open-source, dynamic linker/loader with *glibc* compatibility for the x86_64 architecture supporting automatic and transparent updates of off-the-shelf shared libraries
- Evaluation of popular, binary-distributed shared libraries of recent Debian and Ubuntu releases to assess the live-patching approach's practicality

The remainder of the paper is organized as follows: After giving an overview of DSU techniques in Section 2, Section 3 describes the details of ELF binaries that we utilize for our approach in Section 4. To verify the results, we back-test LUCI with popular shared libraries in Section 5 while classifying the results in Section 6. Section 7 concludes the paper.

2 Related Work

Research of dynamic software updates in user-space dates back four decades, with *DYMOS* [11] presenting the first notable approach beyond manually live-patching machine code. The approach allows updates of programs written in a Modula dialect while using a custom compiler and runtime system. Thereby, this approach involves restrictions with respect to the source-code development and its build system, which correspond to Ⓐ and Ⓒ in Figure 1. Many sophisticated approaches that evolved since then share these restrictions:

Ginseng [28, 29] allows changes in function prototypes and data representation but is limited to source code written in C while also requiring code adjustments Ⓐ. Patches are generated by comparing the source files Ⓑ in conjunction with analysis information emitted by a custom compiler Ⓒ and loaded with a custom runtime system. The approach involves significant performance overhead of up to 30% in updated functions. Although being a powerful approach, *Ginseng* illustrates the massive adjustments required to support generic dynamic updates. LUCI avoids these requirements as it hot-swaps code while not supporting data modification.

With *Kitsune* [15] (successor of *Ekiden* [17]), developers have to manually specify update points, add control-flow and data-migration code to their C source Ⓐ, and use a custom compiler Ⓒ. After waiting for all threads to reach the update point, it replaces the whole program and performs all migrations. The approach causes a service disruption on update time but reduces overhead costs during normal execution.

POLUS [8] constructs a patch for a successive version using a source-to-source compiler Ⓑ. This patch can be applied at any time by employing `ptrace`, with old and new versions residing in memory. *POLUS* places redirection instructions

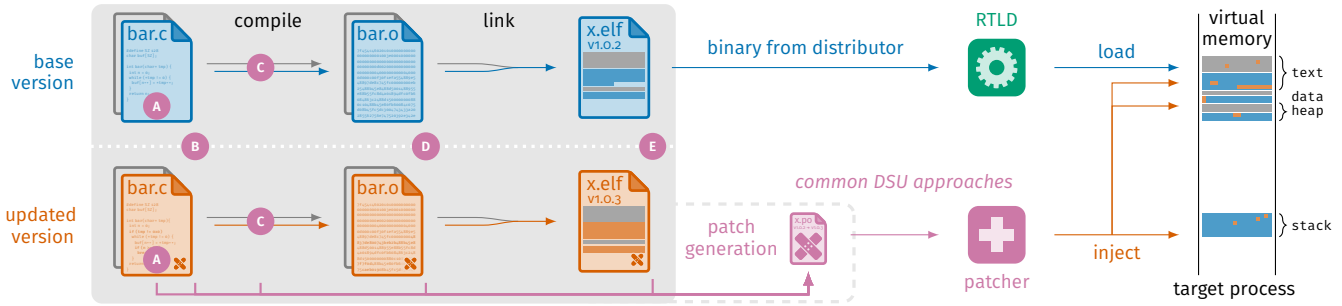


Figure 1: Common DSU approaches require either certain (higher-level) programming languages [11], modification of the code [29], or the changes in the build toolchain [42] (unless the patch is handcrafted from the resulting binaries [5]), as highlighted by the gray area on the left side. Due to the sheer amount of different software projects, it is not feasible to adjust each of them – hence preventing live-patching in user-space for today's real-world software. A generic automatic update mechanism must not interfere with these steps but merely use the resulting binaries, usually built by distributors.

into the old functions and keeps track of global-state modifications. LUCI adapts the idea of having multiple versions of a binary in the virtual memory and extends it by shared data.

In contrast to previous approaches, *Katana* [42] is language-agnostic and works on object-file level (D). While improving applicability, it still requires interfering with the build chain.

For LUCI, the feature of coping with off-the-shelf shared libraries is essential – a feature also provided by a few other approaches: Using dynamic binary translation, *DynSec* [34] can patch code of unmodified binaries while having a significant runtime overhead (11% in benchmarks) and requiring programmer-guided patch generation (E). *Piston* [43] is trying to exploit vulnerabilities in order to fix them. It works on a binary level and can automatically generate repair routines for stack-based buffer overflows, but such routines must be manually provided (E) for other vulnerabilities. With the process virtual machine *DynamoRIO* [5], binary-level code modification (E) is possible at the cost of runtime overhead. Based on this tool, *ClearView* [35] is able to learn normal application behavior and automatically generate patches for certain types of bugs, however, causing massive overheads (depending on the configuration 47% – 303% baseline overhead, and up to 30 000% while learning).

Different methods to update active functions have been proposed: *UpStare* [26] (C) and *StrongUpdate* [51] (B) using stack reconstruction and *ISLUS* [9] (B) with checkpoint-based rollback, all require C source (A). For LUCI, this is not needed since we expect shared library functions to return eventually.

While live updates for Linux [7, 27] and other operating systems [1, 3] were already an important research topic, *kSplice* [2], *kPatch* [38], and *kGraft* [33] (C)(D) initiated kernel live-patching in Linux [20]. The latter two can only perform changes on functions, not on data, similar to LUCI. In contrast, *kSplice* is not only able to support data changes but can also be used for certain specially prepared user-space libraries [31].

A few other approaches focus on live-patching of shared libraries as well: *LibCare* [47] generates patches from assembly emitted by a compiler wrapper script (C) and applies them using `ptrace`: After acquiring storage for changed code (and new data, if required), relocations in the existing code are updated while using stack unwinding to prevent modifications of currently executed functions.

A rather less-intrusive approach is used by *libpulp* [45], requiring specially prepared/compiled libraries with an additional `nop`-prologue at each function (C) to be able to dynamically insert trampoline code. A manually created description file (E) guides the updater (using `ptrace` in conjunction with a preloaded library) through the symbols to be replaced.

The *libDSU* [30] concept also targets unmodified shared libraries. However, *libDSU* would require an actual implementation of the approach to find and update all locations of pointers in the process' memory – including heap chunks, which is quite complex and error-prone to identify. LUCI avoids this requirement by keeping memory locations valid.

All existing approaches either require modifications during building, for both base and updated version, or programmer-guided patch generation. To the best of our knowledge, no approach yet exists that can provide an update on shared libraries without interfering in the build process (gray area in Figure 1). With LUCI, we close this gap.

3 Background

On Unix-based systems, a machine-code–generating compiler produces relocatable objects from the source code, nowadays in the *executable and linkable format* (ELF). With ELF being the fulcrum for executables, LUCI exploits this format.

The ELF meta information lists available and required symbols (e.g., functions and static variables) accompanied by relocation information, enabling the linker to fix destination-address parameters of memory-access and branching instructions in the machine code. For static (position-dependent) executables, the target addresses of all symbols can and must

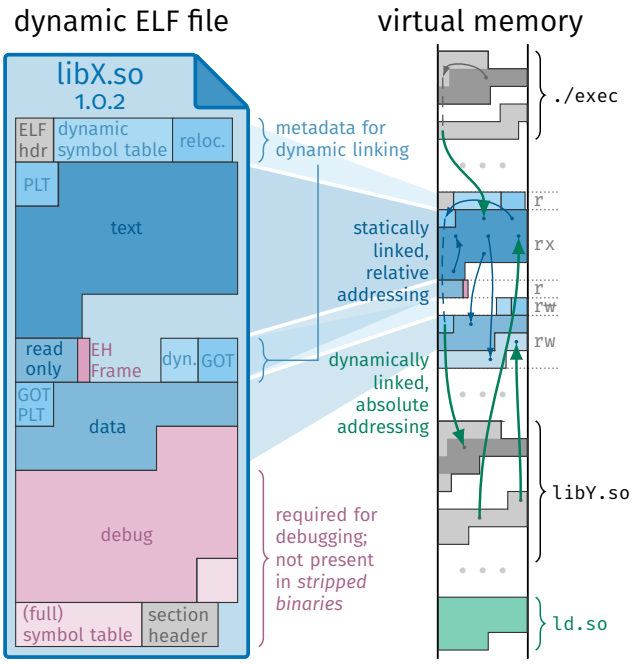


Figure 2: Shared library loaded and relocated in process' virtual memory by the dynamic linker/loader (`ld.so`).

be resolved during static linking, hence, the *executable ELF* contains a complete image of the process's virtual memory. However, for shared objects and dynamic executables, some symbols may remain undefined in the resulting *dynamic ELF* binary. Therefore, the `dynamic` section references their meta information, so symbols can be resolved during execution by the dynamic linker/loader (also referred to as *runtime linker*, RTLD) – allowing the use of other shared libraries.

To simplify the execution of such binaries, the compiler (e.g., with `-fPIC` for position-independent code) references local symbols using relative addressing, while emitting instructions and function stubs (like the *procedure linkage table*, PLT) for indirect addressing to access external symbols. The PLT itself is closely tied to a *global offset table* (GOT) section introduced by the static linker, which stores the actual target addresses during runtime. Therefore, the dynamic linker/loader is not required to modify the machine code in the `text` section itself but only the GOT and, if required, data sections. This strategy improves load performance and security since no executable pages are mapped with write permission in the process' virtual memory, while also maintaining a low memory footprint. Additional techniques like *relocation read-only* (`data.rel.ro`) further contribute to security by removing the write permission after the initial linking steps where applicable (e.g., constants referencing external symbols).

The dynamic linker/loader is responsible for finding all required shared libraries of an application on the file system. The tool places libraries in the process' virtual memory, resolves and fixes undefined symbols while retaining a defined search order (to deterministically handle symbols having the same name). Eventually, the dynamic linker/loader performs initialization and passes control to the application's entry point. For lazy binding (i.e., resolving undefined function symbols on the first call), the dynamic linker/loader has to reside in memory during the entire lifetime of the process. The same holds for the dynamic metadata of each shared library (for symbol lookup and relocation). Figure 2 visualizes a mapping of an ELF file into the virtual memory of a process.

Since a POSIX-conform dynamic linker/loader also provides an interface allowing the process to load additional shared objects at runtime, it is itself an executable, self-contained shared object (`ld.so`) with a tight connection to the C standard library (*libc*) used on the target system. In fact, the dynamic linker/loader is usually an integral part of the standard library (e.g., *glibc*, *musl libc*).

4 Approach

When a modification of a shared object used in a process is detected, we first compare the old and new version. A crucial requirement for our approach is having identical writeable sections: The same symbols have to be stored at the same position having the same size and the same initial content, for both initialized and uninitialized data (`data` and `bss` section). Only then the new version can safely be loaded and linked into the process. We found in practical applications (see Section 5) that this requirement is not a major restriction as modifications on writable sections are rare, as detailed in the following.

While analyzing typical bugfixes for common weaknesses in dominating system programming languages[3], we noticed that newly introduced static variables are rare. Further, modern compilers and linkers work in a deterministic manner (including reordering of variable allocations in an optimization step) and most distributors have optimized their build chains for reproducible builds[4]. Therefore, the probability is high that code changes do not affect the data section of the binary. Consequently, our DSU mechanism checks for alterations in sections containing writeable data, including the *thread-local storage* (TLS), which would prevent an update. While modifications of the writable sections are rare, changes to the read-only data section are more likely, for example, due to introduced strings for error messages. However, these `rodata` updates do not hinder the update process but are inherently supported by our approach, as well as newly introduced automatic variables, which are stored in the stack memory.

[3]The common weakness enumeration (CWE) list at cwe.mitre.org maintains a good overview, including views explicitly focusing on C/C++.

[4]For reproducible builds any indeterminism in the build process is removed, allowing to reproduce a binary-identical file on every build with same source code revision as input. Further information at reproducible-builds.org.

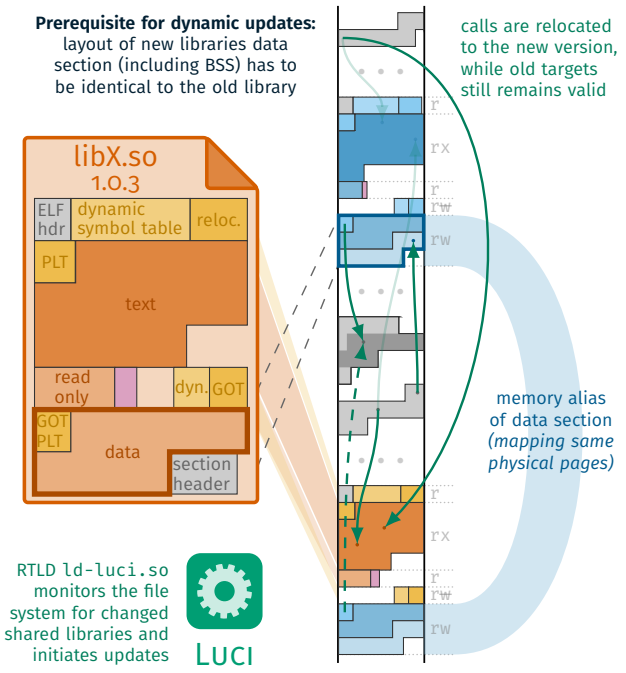


Figure 3: LUCI's core principle is to map the updated shared library to unused space of the processes' virtual memory. Thereby, it exploits *memory aliasing* in order to map the newly introduced data section to the physical memory of the previously active data section.

Segment Layout Requirements In both the old and new version, our approach additionally requires the `data` section to have identical page alignment. However, due to the page-level granularity of permissions in memory-management units, it is already standard in linker scripts to place the writable segment at a page border. The GOT dedicated to the PLT for lazy runtime linking is put at the beginning, followed by the `data` section. The writeable segment is usually preceded by the segment containing the relocation-read-only section, which also includes the standard GOT (for external variables) and is only writeable during the initial dynamic linking stage – but neither its alignment nor its content affects the updateability.

After the updater has ensured the availability of all currently required symbols in the new shared object, it loads all non-writeable data sections into the process' virtual memory, placing it at some previously unused address range. Instead of loading the data section from the ELF file, a *memory alias* of the old version's data section is created (see Figure 3): Both old and new versions' data sections use the same page frames, allowing changes to a variable in the old data section to be immediately visible in the new one and vice versa. Since Linux currently lacks a direct way to create such an alias, we have to use an anonymous in-memory file created by the `memfd_create` system call for the data.

Relinking using GOT Then, all executables and other shared objects utilizing the changed library are updated: The dynamic software updater relocates the affected entries in every GOT to the corresponding symbols in the new library. In addition, it is possible that data sections need relocation as well, for example, when function pointers are used. In this case, the LUCI updater must first ensure that the target memory still contains its original value and is compatible before replacing it with the new value.

Since function calls are performed indirectly (using single instruction reads of the GOT PLT) and both old and new functions coexist in the process, there is no need to alter the `text` section. Hence, the update can be initiated at any time without requiring quiescence [8], reaching a certain update point in code, or modifying the process stack. Furthermore, there are no limitations regarding updates of multithreading applications as there are basically no other runtime modifications than the default RTLD performs when lazy binding. Having the code of an old library function be executed at the time of update is not a problem; it continues accordingly until completed (`return` instruction) with the old code, but future calls will be redirected to the new library function, thus leading to a gradual update. Since shared libraries should provide a collection of subroutines and, therefore, frequently return to the caller, proper software engineering prohibits remaining in an endless loop inside the old version.

In order to apply changes, the dynamic software updater needs access to the process' virtual memory. Since the dynamic linker/loader resides in the process, it is a comfortable target to house the update mechanism, not only avoiding the need for additional permissions but also providing easy access to a list of all loaded objects and their relocation information.

RTLD with DSU Capabilities In order to assess the applicability of the approach to real-world libraries, we have implemented our own dynamic linker/loader LUCI for the x86_64 architecture with the ability to perform the described update procedure while maintaining binary compatibility (to some extent) to the *glibc* equivalent `ld-linux.so`, allowing loading and live-patching of unmodified binaries from a distributor: LUCI can run common executables when passed as a parameter or transparently if LUCI is set as the interpreter in the corresponding ELF section of the executable. This fully circumvents `ld-linux.so` but still supports most *glibc* libraries (including `libc.so` and `libpthread.so`) while providing compatible interfaces for RTLD-specific functionality (e.g., `libdl.so` and tunables).

If dynamic updates are enabled (e.g., by the corresponding environment variable), LUCI creates an observer thread on start and uses the `inotify` API to detect modifications of all loaded shared objects (or their symbolic links, respectively). In case of a `fork`, LUCI intercepts it in order to decouple the data memory alias and create a new observer thread in the child process. Besides the `fork` and thread creation, our

live-updating mechanism introduces no runtime overhead.

On detected changes, the observer thread autonomously compares the update compatibility of both versions and, on success, initiates the update process. In addition, LUCI can output status information, for example, notifying about required restarts due to incompatible changes.

A successful update currently does not necessarily prevent the execution of old versions: If the library dynamically stores a pointer to local functions during runtime (e.g., in heap-allocated memory), these could result in reusing old code after an update. LUCI cannot statically identify (and fix) them since there is no relocation information. Nevertheless, it provides an optional method to detect such access at runtime: After a user-defined time following an update, LUCI hides all pages containing executable sections in old libraries and installs a user-space page-fault handler (using `userfaultfd`). If the non-present page is accessed, LUCI makes it available again while using the status output to inform about the failed update and the requirement for a manual restart.

The virtual memory layout of the updated shared object is as intended by the linker and described inside the ELF file, hence not limiting debugging capabilities with standard tools like *GDB*. Furthermore, C++ exception handling (and unwinding in general) works as expected, even in updated libraries, since LUCI provides a version-agnostic interface for dynamic linker introspection: Therefore, requests from exception/unwinding routines will be passed to the exception handling frame (`eh_frame`) of the corresponding library.

Checking Basic Compatibility To safely create a memory alias of the old writeable data section in an updated shared library, the section must match in alignment, layout, and content. The meta information of the ELF file (`data` section address and size, its initial values plus the dynamic symbol table) can be sufficient – unless the writeable data has local relocations: For each static variable pointing to a local symbol, LUCI ensures the target's equivalency in the old and new version. Otherwise, an update could lead to undesirable behavior, for example, because of changed variable semantics.

The dynamic update is straightforward as long as the target is located in the `data` section, because LUCI simply compares the contents and follows the relocations. However, when considering the executable section, we cannot just compare the machine code since changed offsets (e.g., due to added code) likely result in a different byte stream. With the help of the capstone engine [41], LUCI disassembles the code and (using relocation information, instruction-pointer–relative addressing, and branch instructions) creates a dependency graph. This graph is similar to a call graph but also contains references in the data section. To enable fast comparisons, LUCI calculates a fingerprint for each function: Similar to techniques used in malware analysis [10], LUCI creates a hash based on the machine instructions while excluding relocated immediate operands and `%rip`-displacements – they are replaced instead with corresponding symbolic equivalents. This allows LUCI to find identical functions independently from their location.

For dynamic updates, LUCI considers a target symbol to be compatible if the fingerprint (for symbols in the executable section) or contents (for data) in old and new versions match, as do the targets of all references.

After ensuring the update compatibility in principle, LUCI further checks if the process has not altered the memory targeted by the relocation entry. Consequently, LUCI keeps track of previous values used for fixing relocations and aborts the update process on detected changes while notifying the user about the requirement for a manual restart.

Improving Compatibility Detection So far, stripped binaries – containing only essential parts – are sufficient. However, access to binaries' full symbol table and debug information can contribute to detecting whether an updated version can be applied: While its full symbol table contains storage information for local variables, the *DWARF* debug sections allow an even deeper inspection. Not only type information for all variables can be gathered here, but also the fields of records (`struct`) and enumerated values can be compared.

For debug purposes, many distributors like Debian and Ubuntu offer additional debug information for the binaries, due to size considerations usually distributed in separate packages. With `libdebuginfod`, there is even a web service for easy retrieval of debug symbols, using the unique *BuildID* of the binary located in the `note` section.

Although modification of records (e.g., `unions`) and additional `enum` values do usually not interfere with a successful update, they are still quite rare in non-feature-updates. Thus, we prefer a pessimistic approach for the sake of stability: If either variable location or type, any internal record field, or enumerated value has changed, the update is not applied. This limitation has the ability to simplify the version comparison drastically: By suitably hashing the information about the internal structure, the hash value is sufficient to determine compatibility. We have implemented a service (similar to `libdebuginfod`) that LUCI queries. An alternative to LUCI's approach is to include the value directly in the ELF file, for example, in an additional `note` section inserted by the post-processing steps of the packaging toolchain.

Dynamically Loaded Libraries The POSIX function `dlopen` enables one to load libraries during runtime. By default, both functions and global symbols are retrieved as pointers using `dlsym`. To effectively support dynamic updates for such runtime-loaded files, LUCI creates an indirection for function types: Similar to the PLT, the address of a helper function is returned, which redirects the call to the latest version of the symbol – introducing the overhead of an additional `jmp` instruction. Using this trampoline technique, LUCI can even update dynamically loaded libraries.

5 Evaluation

We successfully validate the previously described functionality of our implementation with a custom test suite written in C/C++ consisting of small examples (targeting especially the corner cases) using different compilers and versions (*GCC* v6 – v12, *LLVM/Clang* v11 – v15) on several distributions (including RHEL/AlmaLinux, Fedora, and openSUSE Leap). Other tests demonstrate the ability to update code changes in libraries written in Ada (*GNAT*), Fortran (*GNU Fortran*), Go (`c-shared` using the *GNU Go Compiler*), Rust (*Rust compiler* with `prefer-dynamic` flag), and, with some limitations, Pascal (*Free Pascal Compiler*).

However, to demonstrate the practicability of our approach, as well as its limitations, this evaluation focuses on popular shared libraries without any custom modifications. To provide a realistic scenario [44], we do not perform updates of individual patches/commits but on the level of full official release versions, which usually contain multiple changes.

For each library, we independently build (without using any artifacts of a previous compilation) each version of a reasonable range having a compatible API. As far as possible, we use the suggested toolchains and default configuration for each library according to the corresponding documentation. Neither changes to the source nor custom tools are used during the build process. In order to evaluate their impact on LUCI's compatibility detection, we manually enable debug symbols.

Then, a program – preferably a test suite with high code coverage – linked against this library interface is executed, while a supervisor script subsequently, with a certain delay, exchanges the library version on the file system in the background. Moreover, this supervisor listens to the status interface of LUCI in order to be notified about incompatible or failed updates (the communication is strictly uni-directional) – which will cause a restart of the test program, enabling it to use the latest library version in the traditional way.

In addition to self-compiled vanilla versions, we also test the corresponding binary releases in popular distributions the same way. We choose Debian and Ubuntu as they are considered to be the most widely used Linux distributions (at least for web servers [48]). Additionally, we are able to retrieve outdated packages from previous releases[5]. We focus on their two most recent versions: *Focal Fossa (20.04)* and *Jammy Jellyfish (22.04)* in the case of Ubuntu (LTS) and *Buster (10)* and *Bullseye (11)* for Debian. However, as debug symbols for some versions are missing in the archives, LUCI solely relies on the (stripped) ELF files when evaluating external builds.

It is worth noting that Debian carefully tries to prevent any breaking changes in their stable package releases, often backporting security fixes to the library version used initially in a particular Debian version. Therefore, its library versions may differ from custom builds having the same version number.

Suitable libraries must meet the following criteria:

- Enough *recent development* to compare different version releases – especially different binary releases in the mentioned distribution versions.
- *Independent libraries* providing distinct functionality rather than just an interface to a service. To simplify testing, it should not be tightly entangled with system components.
- Availability of a *test tool or suite* with reasonable coverage of the library interface and its code. It must not use internals beyond the public/official interface since this might prevent it from running with other releases. To demonstrate the dynamic update, we further need a long-running process (ideally executing the tests in an endless loop) – scripts executing individual tests in subprocesses are unsuitable.
- *High popularity* in both local installations and software depending on it. The Debian popularity contest [40] tracks the installations of their users and can act as an indicator.

Taking those requirements into account, we have selected the libraries *Expat*, *libxcrypt*, *OpenSSL*, and *zlib* for evaluation, all of them within the top 150 packages (out of 70 000) in the Debian popularity contest. This also covers main dependencies of the *nginx* and *Apache HTTP server*.

Environment

We perform all tasks in container environments with a minimal base system installed, running on an x86_64 architecture (Intel Core i5-8400 with four cores and 16 GiB of RAM).

For all tests, LUCI is configured to automatically detect changes in library files or their symbolic links on the file system, check compatibility, and, if applicable, update libraries during runtime. A few seconds after applying an update, the executable section of older library versions is unmapped. Any subsequent access to the old library would trigger the user-space page-fault handler, which marks the update as failed and results in a restart by the supervisor script several seconds later. The delay before a restart allows us to ensure that the program correctly continues even after a failed update, not producing unexpected results or aborting. Incorrect results or abnormal program terminations, regardless of whether the library release or the update causes them, are explicitly noted.

5.1 Expat

Since the *Expat XML parser* is used in numerous applications [46] and hence part of all popular *Linux* distributions, its dozen critical vulnerabilities discovered within the last decade make it a good target and test candidate for LUCI. We focus on major version 2, having 27 version releases since 2006 – with 29 CVEs (including 11 critical vulnerabilities).

[5] Using launchpad.net for Ubuntu and snapshot.debian.org for Debian.

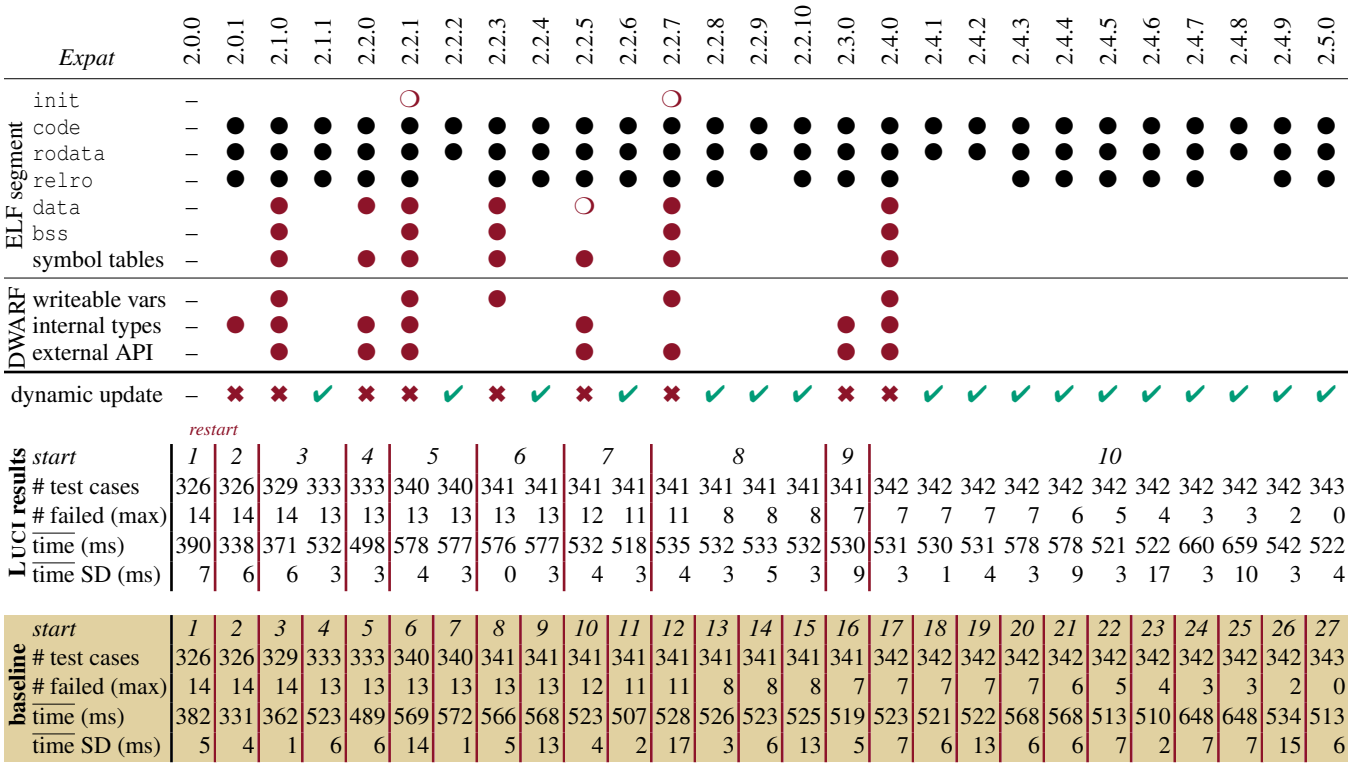


Table 1: Successively (every 25s) replacing the *Expat* library on the filesystem with all vanilla version 2 releases in chronological order while running the test suite (in an endless loop). The upper part of the table shows LUCI's internal analysis of each changed library binary compared to the previously loaded version. ● denotes detected changes of symbols in the corresponding segment, while ❍ marks identical segments having symbols with modifications in their dependencies (at other segments). When non-updatable changes (highlighted with red color) are detected, the library version is incompatible (✖) and the test suite gets restarted (marked by a vertical bar: |). Compared to the baseline (using the default RTLD, shown on the bottom), LUCI can prevent two thirds of restarts while providing the same results (as shown in the middle row).

The developers maintain a good and regularly updated test suite in a single program, which we have to slightly modify: Tests causing segmentation faults and double-frees in older releases are dynamically omitted in those vulnerable versions. Further, due to a slight API change (new symbols in 2.1.0 and 2.4.0), the corresponding tests are only enabled if those symbols are available in the currently active library release (using weak linkage). The LUCI-loaded program now executes all eligible tests sequentially in an endless loop while measuring the duration and the number of executed and failed tests[6].

Vanilla The results in Table 1 show that LUCI is able to perform 17 dynamic updates (67%) during runtime – with 11 subsequent updates (starting with version 2.4.0) without requiring any restart. During those patches, we observe a steady decrease in failed test cases due to the bugs fixed in newer releases, identically to manually executing the test suite with the corresponding library version using the default RTLD.

[6]A use-after-free bug (CVE-2022-40674) causes jitter in the results for versions prior to 2.4.9, as the corresponding test only sometimes fails.

The average duration of a test iteration in LUCI is slightly worse (about 2%), but this is caused by our RTLD implementation itself since it is not as optimized as the *glibc* counterpart and needs to use some workarounds/indirections for compatibility with standard `libc.so.6`: The timings with LUCI are consistent regardless of whether the DSU functionality is enabled or disabled. Furthermore, the raw data does not show any notable increase while updating a library to a new version.

Most failed updates have obvious reasons, like changes in the writeable data section, which LUCI automatically detects in both binary and DWARF debug information. However, LUCI rejects the updates to 2.0.1 and 2.3.0 only due to debug information. A detailed look reveals that in both cases only a single enumeration value was added. This causes a different hash of the datatype (and, in the latter case, the API as well) even though an update would be possible – which we validate in additional tests. However, the currently strict setting safely allows certain update directions: It is not only possible to skip several releases (e.g., 2.4.4 → 2.4.9), but rolling back to an earlier release is also possible – as long as the hashes are identical and the requirements of the binary are met.

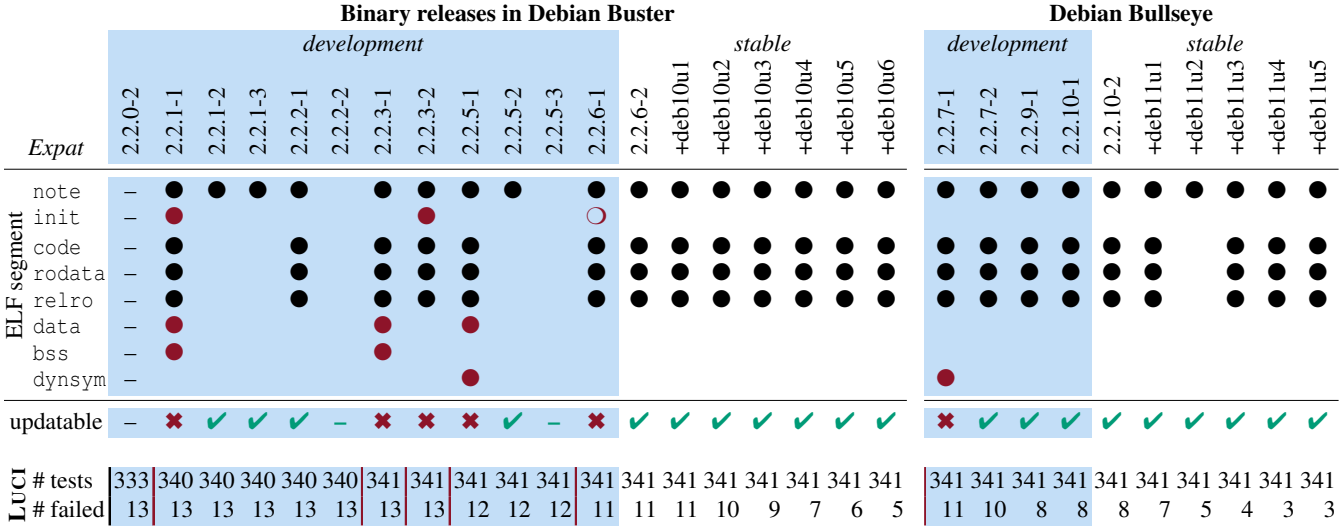


Table 2: *Expat* test suite running with binaries retrieved from the official Debian repository – including development builds.

Backtesting Additionally, we back-test the approach using prebuilt packages from distributors. For Debian, we can retrieve the binaries from the snapshot archive, which is not limited to updates for stable releases but includes all development builds as well. The implications become apparent when considering the Debian workflow: Whereas during the development phase latest library versions are maintained – having the same update incompatibilities as our vanilla builds above – the versions are frozen after the Debian release gets stable. However, that does not mean a standstill at all: The Debian team puts effort into backporting fixes for issues, like the decreasing number of failed test cases in Table 2 shows: While *Debian Buster* uses *Expat* 2.2.6 in its stable release 2019, their latest package `2.2.6-2+deb10u6` fixes several vulnerabilities found in 2022. Since they usually do not include feature changes but only minor changes in the code section, the stable phase is an ideal situation for LUCI: all package releases are eligible for dynamic updates. When also considering development builds, LUCI can prevent 72% of restarts in Buster and 90% in Bullseye.

Although the stable phase used in productive environments is our main focus for LUCI, the development builds offer interesting insights about the applicability of our approach: Several different versions of the build utils were used during that time; however, this does not necessarily cause incompatibilities. For example, package 2.2.9-1 was built with *GCC* 9.2, while 2.2.10-1 used *GCC* 10.2, but a dynamic update is still possible. It can also happen that a newer release build results in identical code and data segments, which may or may not produce a different *BuildID* stored in the `note` section. For example, the unequal *BuildID* between 2.2.5-1 and 2.2.5-2 requires LUCI to analyze the file, while the subsequent update to 2.2.5-3 is binary identical, allowing LUCI to safely skip any further processing quite early.

Ubuntu is similar regarding the version freeze after stable release, as shown in Table 3: Even with development libraries, 83% of restarts can be omitted with LUCI in *Jammy*. Library updates published for the stable release can all be dynamically applied; the same is true for all library releases in *Focal*. In both versions, there is only one update each, simply differing in the *BuildID* compared to its predecessor; all other updates have actual code changes.

These results highlight the effectiveness of LUCI in a real-world scenario, as it can update the majority of all *Expat* versions, even off-the-shelf libraries built with different compilers and without access to debug information.

Build		*Expat*	updatable
custom (vanilla)		2.0.0 – 2.5.0	17 / 26 (65%)
Debian Buster	*all*	2.0.0 – 2.2.6	13 / 18 (72%)
	stable	2.2.6	6 / 6 (100%)
Debian Bullseye	*all*	2.2.7 – 2.2.10	9 / 10 (90%)
	stable	2.2.10	5 / 5 (100%)
Ubuntu Focal	*all*	2.2.7 – 2.2.9	6 / 6 (100%)
	stable	2.2.9	4 / 4 (100%)
Ubuntu Jammy	*all*	2.4.1 – 2.4.7	10 / 12 (83%)
	stable	2.4.7	2 / 2 (100%)

Table 3: Summary of LUCI executing the *Expat* test suite using different library builds (including off-the-shelf binaries).

5.2 libxcrypt

The *extended crypt library* [14] is a modern replacement for *glibc* `libcrypt.so.1`, providing various one-way hashing methods that are frequently used for user authentication. Several test cases are included in the source. Most of them solely

Build	*libxcrypt*		updatable		
custom (vanilla)	4.0.0 – 4.4.33	*all*	35 / 47	(74%)
		unqiue	19 / 31	(61%)
Debian Bullseye	4.4.10 – 4.4.18	*all*	7 / 7	(100%)
		unqiue	3 / 3	(100%)
Ubuntu Focal	4.4.10	*all*	8 / 8	(100%)
		unqiue	0	–
Ubuntu Jammy	4.4.18 – 4.4.27	*all*	4 / 4	(100%)
		unqiue	4 / 4	(100%)

Table 4: Program running all *libxcrypt* tests in parallel (using multithreading) on LUCI while exchanging library builds.

use the shared library interface without any knowledge about the internal structure and are therefore suited for our evaluation. However, we have to exclude three test cases because of severe memory leaks and division-by-zero bugs in conjunction with older releases. Our test program endlessly repeats each eligible test case in a distinct thread without any synchronization whatsoever, requiring LUCI to apply updates to a process with several active threads.

When building and testing all 48 releases of version 4, LUCI is able to dynamically update 74% of them. However, 11 builds are binary identical to the previous one (e.g., 4.4.28), and 5 builds only differed in their *BuildID* – still enabling LUCI to update 19 out of 32 unique builds with actual code changes during runtime.

During updates, we also observe a steady decrease in failing tests: While 9 (out of 25) test cases report errors when running with the first release of the library (version 4.0.0), there are no more unsuccessful tests after the latest update.

While *Debian Buster* still retains the *glibc* crypt library, *Bullseye* moved to this replacement library (package name `libcrypt1`), which can be fully dynamically updated as stated in Table 4. In *Ubuntu Focal*, one can find 9 different packages, but their code and data are all identical (having 5 different *BuildID*s). In contrast, *Jammy* has 5 actual different builds that are all compatible. Due to the fact that *Bullseye* and *Jammy* only have a single *stable* release, we consider *all* releases, including *development*.

The approach of LUCI does not restrict the update of a library concurrently employed by several dozen active threads in a multithreaded application, as these results show.

5.3 OpenSSL

Because of its broad application – and several severe vulnerabilities in recent years – the secure communication library *OpenSSL* has achieved a certain degree of brand awareness. We focus on its two main libraries `libssl.so` and `libcrypto.so` and set up a client-server environment using the `openssl` utility for testing.

Of 20 releases in OpenSSL 1.1.1, only 6 versions of `libssl.so` seem to be updatable and none of `libcrypto.so`. Further investigation showed that the `ssl3_undef_enc_method`-structure could be blamed: Its members point to functions that can reach `ERR_raise` using an array containing error messages, which are frequently edited in the source. This would not be a problem if the structure in question, which is – to the best of our knowledge – never modified, were marked as constant. However, since it is currently writeable and hence placed in the `.data` section, LUCI requires all of its references recursively to be identical to the previously loaded library for updates. Although LUCI's decision to reject such changes works just as intended, when temporarily relaxing this constraint, 9 releases in both libraries meet the requirements for an update.

However, after every update during testing, LUCI detects code usage in superseded libraries and hence reports it as failed. Again, the reasons for this shortcoming are function pointers in writable data: Instead of statically initializing a variable, *OpenSSL* does this during runtime[7] – taking over the work originally intended for the RTLD and hence leaving LUCI with no clue about those relocations, unable to correct them to the updated version.

Using `uprobes`, we can verify that only old code referring to unchanged functions was executed and the updates are effectively applied. However, LUCI is currently not meant to handle such code as further discussed in Section 6.

5.4 zlib

The library *zlib* is used in many software dealing with data compression and was not exempted from serious vulnerabilities. We are testing version 1.2 – since the version numbering switches between three and four places, there are 49 releases from 1.2.0 to 1.2.13 (the latest at the time of writing). Since the full code coverage test of the inflate algorithm cannot be used due to its interference with internal structures, we run the various de- and inflation tests of the *zlib* example file in an endless loop while updating the libraries.

Of all releases in the past two decades, only half are suited for dynamic updates, as the results in Table 5 show. The main reasons preventing an update are the 19 interface extensions during that time, which are often accompanied by additional data structures – both creating different debug hashes. The binary builds show an ambivalent picture as well: due to the interface changes and the fix for CVE-2018-25032 modifying internal structures, none of *Debian Buster*'s builds can be updated. The same CVE applies to *Debian Bullseye* and *Ubuntu Focal*, which have around half their packages eligible for dynamic updates. Only for *Ubuntu Jammy* (whose development began after discovering this CVE), all releases can be applied dynamically.

[7] For example, `names_lh` in `crypto/objects/o_names.c` is initialized with `NULL` and assigned once with a fixed value in a custom `RUN_ONCE` function during start.

Build	*zlib*	updatable		
custom (vanilla)	1.2.0 – 1.2.13	*all*	24 / 48	(50%)
		unqiue	24 / 48	(50%)
Debian Buster	1.2.8 – 1.2.11	*all*	1 / 3	(33%)
		unqiue	1 / 3	(33%)
Debian Bullseye	1.2.11	*all*	4 / 5	80%)
		unqiue	3 / 4	(75%)
Ubuntu Focal	1.2.11	*all*	5 / 7	(71%)
		unqiue	2 / 4	(50%)
Ubuntu Jammy	1.2.11	*all*	4 / 4	(100%)
		unqiue	2 / 2	(100%)

Table 5: *zlib* de- and inflations tests using LUCI while exchanging library builds.

6 Discussion

Unlike several other DSU approaches, LUCI does not aim for general updateability and deliberately sacrifices some features: Most notably, the requirement for identical writeable data segments, which allows multiple library versions to coexist concurrently in memory, prevents changes of static variables. In case the library maintains some sort of state, it must not differ between the old and new versions for the same reason. Hence, changes to the initialization functions are prohibited. Consequently, record types have to be equal as well since they might be used in a shared state during the update transition (e.g., `structs` in heap or stack memory). If the lifetime of a pointer to an internal function outlasts the execution time of the function in which it was assigned, old code may be executed after an update. Furthermore, a related case can occur when a library function resides for a long time on the call stack, possibly because of an endless loop.

Accordingly, LUCI statically verifies the compatibility of data and interface before starting the update, while afterward dynamically detecting the execution of old code: To prevent abnormal terminations, LUCI takes a strict course, favoring false positives over false negatives. If a new version of a library does not meet all requirements and therefore is not eligible for dynamic updates, the process remains running in a valid state while LUCI notifies users about the non-updatable version, so they can manually restart the service. However, in case any subsequent update is again compatible with the currently active library, the update will be applied.

For many libraries, this pessimistic approach is sufficient to update most library versions, especially when it comes to minor (patch-level) changes like stable release branches of distributions: The approach's restrictions rarely apply to bugfixes, which most frequently require a timely deployment. The decreasing number of failures in the *Expat* test suite in Section 5.1 demonstrates the immediate effect of bugfix updates. However, extensive bugfixes and feature changes with cross-cutting changes cannot be applied, as seen in major version updates (e.g., *Expat* 2.3.0 → 2.4.0).

A notable exemption is *OpenSSL*: Although it is an interesting target for DSU due to its wide distribution, it is notorious for its code quality [4, 19], and hence the failing results are not surprising. While a first examination suggests that a few changes in its code (which we rather categorize as coding-standard fixes) would considerably improve its updateability, there are possibilities for LUCI to handle such libraries: By using a fine-grained code-access–detection method like `uprobe` and `ptrace`, further conclusions about the actually executed symbols in old libraries can be drawn. Accordingly, LUCI can ignore accessed symbols that are identical to their corresponding newer version. However, in contrast to the currently employed coarse-grained user-space page-fault mechanism, this would require additional permissions and induces overhead. Nevertheless, if a function pointer actually refers to a symbol that has been modified in the updated library, countermeasures are possible: LUCI could alter the old library's code in such a way that it redirects the control flow to the corresponding new version of the symbol (e.g., by using a `nop` slide) – but at the same time losing the advantage of simplicity and safety while not having to modify the `text` section.

It is worth mentioning that function pointers to symbols in other shared libraries do not point directly to the target but to the corresponding PLT entry – which LUCI fixes on update. Moreover, the use of virtual inheritance in C++ is not affected as well: Such objects are internally extended by an additional `vpointer` attribute, which references the `vtable` stored in the (relocation-)read-only section and is updatable.

A crucial limitation that LUCI shares with other automatic DSU approaches is the application of structural changes, which can become arbitrarily complex due to the semantic gap between (the intentions at) the source code level and the information available at the binary level after compilation. They are generally unrecognizable on a binary level without further context.

Listing 1 shows a contrived example of a change in a spinlock implementation: The semantic change of the value signaling the holding of a lock from `1` to `0` is only reflected in

```
1  typedef int lock_t;
2
3  void lock(volatile lock_t * var) {
4  -     while(!CAS(var, 0, 1)) {}
5  +     while(!CAS(var, 1, 0)) {}
6  }
7
8  void unlock(volatile lock_t * var) {
9  -     *var = 0;
10 +     *var = 1;
11 }
```

Listing 1: A contrived example of a code modification that LUCI cannot automatically reject, due to changes at a higher semantic level.

the functions' machine code. It is indistinguishable from a valid (i.e., bugfix) code change, and it does not change the interface. Consequently, LUCI cannot automatically detect this as an incompatible update. Even restrictions such as *activeness safety* [16], which prevents active functions (on the call stack) from updating, would allow an update point inside the critical section: For example, if one thread can hold the `lock` and the incorrect update is applied, then another thread can incorrectly acquire the `lock` a second time. However, proper software-engineering techniques requiring `lock_t` to be an `enum` with constants `LOCKED` and `UNLOCKED` would enable LUCI to detect the incompatibility using *DWARF* information and reject the update.

Data structures can pose similar issues regarding the detection of incompatible changes: While changes in `structs` are reflected in the debug information, a semantic modification of the access using only pointer arithmetic and casting is not detectable by LUCI. This lack of further information about the data layout is especially true for dynamically allocated memory. Further problems may arise if the modification of function parameters can have an impact on the process environment (e.g., adding a write-protection flag in `mmap`).

The problems described can be tackled with carefully chosen manual update points and hand-crafted (or test-cases–assisted [25]) state transformations. However, this would involve a significant amount of work for each library. For LUCI, we instead propose a more pragmatic solution by analyzing the change set at the source code level and explicitly marking a binary as *compatible* or *incompatible* with the previous version. Maintainers or distributors usually have the knowledge to perform this compatibility review. The resulting information can be included in the metadata of the binary (`note` section) or the package. Alternatively, the compatibility check can be carried out independently by third parties (e.g., by providing the information along with the debug hash).

To facilitate the compatibility review, LUCI has tooling support to automatically identify obviously incompatible versions (e.g., different writeable section). A further LUCI tool for simplifying the review shows the associated source-code lines that belong to the modified code in the resulting binary (using the *DWARF* symbols). With LUCI, we argue that less knowledge is required to decide on the update compatibility compared to manually writing update routines (e.g., introducing update points or writing state transformers).

In summary, Listing 1 illustrates that corner cases exist that circumvent LUCI's automatic compatibility check. Therefore, as mentioned, LUCI's tooling infrastructure assists the user in performing manual compatibility checks. However, all the libraries we have analyzed so far have well-structured code that does not require manual intervention. From this practical observation, we argue that LUCI's approach solves numerous real-world code-patching problems.

Regarding the aspect of LUCI's memory demand, we argue that having multiple coexisting versions of the library in memory is rather unproblematic: Non-writable segments are file mappings and, therefore, do not permanently reside in memory. The data segments exist only once due to LUCI's memory-aliasing technique.

7 Conclusion

Even though dynamic software updates are a well-received research topic and the benefit due to security concerns is undisputed, they barely made their way to user space on our everyday systems due to the required effort for software developers. In this paper, we propose a concept addressing the existing obstacles, hereby focusing on the most frequently reused kind of software: shared libraries.

Analysis of common bugfix patterns, including their effect on the resulting ELF files, allows the conclusion that the metadata is sufficient to enable a dynamic linker/loader to update today's binary-distributed shared libraries without requiring any changes or inducing additional runtime overhead.

To validate this claim, we have implemented our own dynamic linker/loader LUCI, which acts as an evaluation platform for live-patching common binaries: LUCI dynamically updates many versions of popular shared libraries like *Expat*, *libxcrypt*, and *zlib*. But even in the case of incompatibilities, normal execution is maintained: In no case does an update, neither successful nor failed, lead to an abnormal program termination or incorrect behavior during our evaluation.

The presented results suggest that the proposed approach is practicable and can play a part in paving the way for the common use of live-patching of user-space applications.

Although binary-compatible and supporting several of its interfaces, LUCI is not deemed a replacement for the existing default dynamic linker/loader (e.g., from the *glibc* project) but is intended to support further investigation and research for loader-based dynamic software updates. We hope that LUCI will help DSU become a reality in user space – and, for example, be supported by the standard RTLD some day.

Acknowledgments

The author order corresponds to the SDC (*sequence determines credit*) model. This work is part of a project campaign on *dynamic operating-system specialization* (DOSS) and is also funded in parts by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 465958100 and 502947440. We are grateful to the anonymous reviewers and our shepherd for their insightful comments.

Availability

The source code of LUCI and its components is published under the *GNU Affero General Public License, Version 3* at github.com/luci-project/luci.

A Artifact Appendix

Abstract

Our artifact includes the source of the dynamic linker/loader LUCI itself, alongside with a script to evaluate its live-update capabilities on the libraries *Expat*, *libxcrypt*, *OpenSSL*, and *zlib* using suitable tests. For building and testing, container environments (based on *Docker*) are employed. The artifact contains scripts to support automatically building these libraries from their official source and tools to download the corresponding Debian and Ubuntu packages. LUCI is intended for a recent Linux environment on an x86_64 architecture.

Scope

We aim to achieve two main goals with the artifact: Firstly, we want to encourage reproducing the results presented in this paper, the artifact therefore supports

- building the dynamic linker/loader LUCI, which is implemented according to Section 4,
- building several release versions of the libraries *Expat*, *libxcrypt*, *OpenSSL*, and *zlib* using the source from official repositories,
- validating the changes of the test suites, and
- running all experiments described in Section 5. This enables one to reproduce the results referred to in text and listed in further detail in Table 1, Table 2 and Table 3 for *Expat*, Table 4 for *libxcrypt*, and Table 5 for *zlib*.

Secondly, since we are aware that there is a lack of "hackable" dynamic linker/loaders (especially when it comes to *glibc*-compatibility), we provide LUCI for academic purposes, mainly but not limited to research on loader-based DSUs.

Contents

The dynamic linker/loader consists of the following internal sub-projects (each distributed in a separate repository):

`dlh` provides basic functionality similar to libc/STL for creating static freestanding applications (without *glibc*).

`elfo` is a lightweight parser for the Executable and Linking Format, supporting common GNU/Linux extensions.

`bean` binary explorer/analyzer to compare shared libraries and detect changes, which uses the Capstone Engine.

`luci` dynamic linker/loader with DSU capabilities and *glibc* compatibility (`ld-linux-x86-64`).

To build LUCI, it is sufficient to recursively clone the repository with its submodules and run `make` in the main folder. Further details are provided in the `README.md`.

For each shared library used in Section 5, there is a corresponding subfolder in the evaluation repository. With `gen-lib.sh`, the desired version(s) are built, `gen-test.sh` compiles the test program (located in `src-test`), and `run-test.sh` runs the experiments with automatic library exchanging in a containerized environment.

Hosting

Both LUCI's source code and the evaluation environment are available at github.com/luci-project/eval-atc23.

The utilities for building the shared libraries retrieve the source code from the following official repositories:

- github.com/libexpat/libexpat
- github.com/besser82/libxcrypt
- git.openssl.org
- github.com/madler/zlib

To acquire the Debian and Ubuntu packages released for each library, the utilities use the web services launchpad.net, metasnap.debian.net, and snapshot.debian.org.

Requirements

We have written all parts of the dynamic linker/loader in C/C++. A standard GCC (version 9 & 10) is sufficient to compile the project. While LUCI has no further restrictions on its build environment, its execution is currently limited to distinct versions of certain distributions, since LUCI must conform to the corresponding *glibc* interface (see Table 6).

To enable all features of LUCI, a Linux kernel 4.11 or newer with a default configuration is required. We recommend a Debian Bullseye installation using its standard kernel image.

We use *Python 3*, *GNU make*, and *Bash* for helper scripts. The tests are executed in a *Docker* container using the official Debian and Ubuntu images. The hardware platform should be an x86_64 architecture with at least 16 GiB of RAM and 6 GiB of storage.

Distribution	Release	glibc
Debian	Stretch (9)	2.24
	Buster (10)	2.31
	Bullseye (11)	2.31
	Bookworm (12)	2.36
Ubuntu	Focal Fossa (20.04)	2.31
	Jammy Jellyfish (22.04)	2.35
AlmaLinux Oracle Linux RedHat Enterprise Linux	9	2.28
Fedora	36	2.35
	37	2.36
openSUSE Leap	15	2.31

Table 6: Distributions currently supported by LUCI.

References

[1] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc A. Auslander, David Edelsohn, Benjamin Gamsa, Gregory R. Ganger, Paul E. McKenney, Michal Ostrowski, Bryan S. Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, 2003. DOI: 10.1147/sj.421.0060

[2] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, pages 187–198, 2009. DOI: 10.1145/1519065.1519085

[3] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference (ATC '05)*, pages 279–291, 2005.

[4] Bob Beck. LibreSSL - an OpenSSL replacement. Berkeley System Distribution (BSD), Andrea Ross, 2014. DOI: 10.5446/15347

[5] Derek Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[6] Matthieu Caneill, Daniel M. Germán, and Stefano Zacchiroli. The debsources dataset: two decades of free and open source software. *Empirical Software Engineering*, 22(3):1405–1437, 2017. DOI: 10.1007/s10664-016-9461-5

[7] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 35–44, 2006. DOI: 10.1145/1134760.1134767

[8] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 271–281, 2007. DOI: 10.1109/ICSE.2007.65

[9] Zhikun Chen and Weizhong Qiang. ISLUS: an immediate and safe live update system for C program. In *Proceedings of the 2nd International Conference on Data Science in Cyberspace (DSC '17)*, pages 267–274, 2017. DOI: 10.1109/DSC.2017.50

[10] Cory Cohen and Jeffrey S Havrilla. Function hashing for malicious code analysis. *CERT Research Annual Report*, pages 26–29, 2009.

[11] Robert P. Cook and Insup Lee. DYMOS: a dynamic modification system. In *Proceedings of the Symposium on High-level debugging (SIGSOFT '83)*, pages 201–202, 1983. DOI: 10.1145/1006147.1006188

[12] Christophe Cérin, Camille Coti, Pierre Delort, Felipe Diaz, Maurice Gagnaire, Marija Mijic, Quentin Gaumer, Nicolas Guillaume, Jonathan Le Lous, Stephane Lubiarz, Jean-Luc Raffaelli, Kazuhiko Shiozaki, Herve Schauer, Laurent Smets, Jean-Paul Seguin, and Alexandrine Ville. Downtime statistics of current cloud solutions. *The International Working Group on Cloud Computing Resiliency*, 2014.

[13] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the Conference on Internet Measurement Conference (IMC '14)*, pages 475–488, 2014. DOI: 10.1145/2663716.2663755

[14] Björn Esser and Zack Weinberg. libxcrypt – extended crypt library for descrypt, md5crypt, bcrypt, and others. https://github.com/besser82/libxcrypt, 2021.

[15] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM Transactions on Programming Languages and Systems*, 36(4), 2014. DOI: 10.1145/2629460

[16] Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Transactions on Software Engineering*, 38(6):1340–1354, 2012. DOI: 10.1109/TSE.2011.101

[17] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State transfer for clear and efficient runtime updates. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering (ICDE '11)*, pages 179–184, 2011. DOI: 10.1109/ICDEW.2011.5767632

[18] Haegeon Jeong, Jeanseong Baik, and Kyungtae Kang. Functional level hot-patching platform for executable and linkable format binaries. In *Proceedings of the International Conference on Systems, Man, and Cybernetics (SMC '17)*, pages 489–494, 2017. DOI: 10.1109/SMC.2017.8122653

[19] Poul-Henning Kamp. Please put OpenSSL out of its misery: OpenSSL must die, for it will never get any better. *Queue*, 12(3):20–23, 2014. DOI: 10.1145/2602649.2602816

[20] The kernel development community. The Linux kernel: Livepatch. https://docs.kernel.org/livepatch/livepatch.html, 2022.

[21] Raula Gaikovina Kula, Daniel M. Germán, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23:1–34, 2018. DOI: 10.1007/s10664-017-9521-5

[22] Andrew Lerner. The cost of downtime. https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/, 2014.

[23] Netcraft Ltd. December 2022 web server survey. https://news.netcraft.com/archives/2022/11/10/november-2022-web-server-survey.html, 2022.

[24] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. SPIDER: Enabling fast patch propagation in related software repositories. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '20)*, pages 1562–1579, 2020. DOI: 10.1109/SP40000.2020.00038

[25] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. *ACM SIGPLAN Notices*, 47(10):265–280, 2012. DOI: 10.1145/2398857.2384636

[26] Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the Conference on USENIX Annual Technical Conference (ATC '09)*, pages 1–14, 2009.

[27] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, pages 327–340, 2007. DOI: 10.1145/1272996.1273031

[28] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 13–24, 2009. DOI: 10.1145/1542476.1542479

[29] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 72–83, 2006. DOI: 10.1145/1133981.1133991

[30] Martin Alexander Neumann, Christoph Tobias Bach, Stefan Kratochwil, Marcel Kost, and Michael Beigl. libDSU: Towards hot-swapping dynamically linked libraries on stock Linux. In *Companion Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '16)*, pages 41–42, 2016. DOI: 10.1145/2984043.2989223

[31] Oracle. New userspace patching with oracle ksplice! https://blogs.oracle.com/linux/post/new-userspace-patching-with-oracle-ksplice, 2015.

[32] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009. DOI: 10.1007/s10664-008-9077-5

[33] Vojtěch Pavlík. kgraft: Live kernel patching. https://www.suse.com/c/kgraft-live-kernel-patching/, 2014.

[34] Mathias Payer, Boris Bluntschli, and Thomas R. Gross. DynSec: On-the-fly code rewriting and repair. In Cristian Cadar and Jeff Foster, editors, *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades (HotSWUp '13)*, 2013.

[35] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09)*, pages 87–102, 2009. DOI: 10.1145/1629575.1629585

[36] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, pages 573–585, 2019. DOI: 10.1145/3297858.3304063

[37] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a stock JVM. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, pages 103–119, 2014. DOI: 10.1145/2660193.2660220

[38] Josh Poimboeuf. Introducing kpatch: Dynamic kernel patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching, 2014.

[39] Debian Project. Debian security FAQ: Why are you fiddling with an old version of that package? https://www.debian.org/security/faq.en.html#oldversion, 2021.

[40] Debian Project. Debian popularity contest. https://popcon.debian.org/stable/main/by_inst, 2022.

[41] Nguyen Anh Quynh, Tan Sheng Di, Ben Nagy, and Dang Hoang Vu. Capstone engine. https://github.com/capstone-engine/capstone, 2021.

[42] Ashwin Ramaswamy, Sergey Bratus, Sean W. Smith, and Michael E. Locasto. Katana: A hot patching framework for ELF executables. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES '10)*, pages 507–512, 2010. DOI: 10.1109/ARES.2010.112

[43] Christopher Salls, Yan Shoshitaishvili, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Piston: Uncooperative remote runtime patching. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*, pages 141–153, 2017. DOI: 10.1145/3134600.3134611

[44] Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. Towards standardized benchmarks for dynamic software updating systems. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp '12)*, pages 11–15, 2012. DOI: 10.1109/HotSWUp.2012.6226609

[45] SUSE. Libpulp. https://github.com/SUSE/libpulp, 2022.

[46] Expat team. Expat XML parser: Software using Expat. https://libexpat.github.io/doc/users/, 2022.

[47] TuxCare. Libcare – patch userspace code on live processes. https://github.com/cloudlinux/libcare, 2020.

[48] W3Techs. Usage statistics of Linux for websites. https://w3techs.com/technologies/details/os-linux, 2022.

[49] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying reuse of out-dated third-party code in open source projects. *Information and Media Technologies*, 9(2):155–161, 2014. DOI: 10.11185/imt.9.155

[50] Asimina Zaimi, Apostolos Ampatzoglou, Noni Triantafyllidou, Alexander Chatzigeorgiou, Androklis Mavridis, Theodore Chaikalis, Ignatios Deligiannis, Panagiotis Sfetsos, and Ioannis Stamelos. An empirical study on the reuse of third-party libraries in open-source software development. In *Proceedings of the 7th Balkan Conference on Informatics (BCI '15)*, 2015. DOI: 10.1145/2801081.2801087

[51] Deqing Zou, Hao Wang, and Hai Jin. StrongUpdate: An immediate dynamic software update system for multi-threaded applications. In *Human Centered Computing*, pages 365–379, 2015. DOI: 10.1007/978-3-319-15554-8_30

MELF: Multivariant Executables for a Heterogeneous World

Dominik Töllner
Leibniz Universität Hannover

Christian Dietrich
Hamburg University of Technology

Illia Ostapyshyn
Leibniz Universität Hannover

Florian Rommel
Leibniz Universität Hannover

Daniel Lohmann
Leibniz Universität Hannover

Abstract

Compilers today provide a plethora of options to optimize and instrument the code for specific processor extensions, safety features and compatibility settings. Application programmers often provide further instrumented variants of their code for similar purposes, controlled again at compile-time by means of preprocessor macros and dead-code elimination. However, the global once-for-all character of compile-time decisions regarding performance-, debugging-, and safety/security-critical features limits their usefulness in heterogeneous execution settings, where available processor features or security requirements may evolve over time or even differ on a per-client level.

Our *Multivariant ELF (MELF)* approach makes it possible to provide multiple per-function compile-time variants within the same binary and flexibly switch between them at run-time, optionally on a per-thread granularity. As MELFs are implemented on binary level (linker, loader), they do not depend on specific language features or compilers and can be easily applied to existing projects. In our case studies with SQLite, memcached, MariaDB and a benchmark for heterogeneous architectures with overlapping ISAs, we show how MELFs can be employed to provide per-client performance isolation of expensive compile-time security or debugging features and adapt to extended instruction sets, when they are actually available.

1 Introduction

Modern compilers provide a plethora of options to statically optimize and instrument the code. Natural examples for such at-compile-time tailoring include support for hardware-specific processor extensions, but also compiler-specific debugging, program instrumentation, and sanitizing aid. These options commonly do not alter the semantics of the code, but influence its *nonfunctional* properties with respect to performance, safety, security, and compatibility. They are put under the control of the developer, because they reflect important tradeoffs: Exploiting special instruction-set extensions can greatly improve performance [1], but at the cost of losing compatibility to smaller or older processors. Letting the compiler instrument the code with extra sanity checks increases safety and security [2]–[6], but comes at a significant performance cost. This also holds for many higher-level instrumentations that are inserted manually by the developers, often by means of the preprocessor: Typical examples are executable asserts [7], [8], tracing, and logging support, which have been shown to actively increase safety [9] and security [10], but are commonly disabled at compile time in production builds due to their performance overhead.

However, in a world of increasing dynamic hardware and use-case heterogeneity, the global once-for-all character of compile-time decisions regarding performance-, debugging-, and safety/security-critical features limits their usefulness: In heterogeneous cloud environments or on machines with heterogeneous ISAs, the availability of processor features may change over time, even for individual threads. The performance costs of the extra sanity checks might be acceptable while processing input from external users, but not in general. In a DevOps setting, it would be useful to temporarily enable tracing and logging, but only for that specific client who is having troubles.

In short: It would be useful to decide at run time, depending on the dynamic context, on features that are technically bound at the compile time of the code. We call this flexibility *semi-dynamic variability*, which conceptually lays between static and dynamic variability in that the code of all variants is still generated at the compile-time of the project (thus, facilitating whole-program optimization), but the actually used variants can be decided on at run time.

While there are special-purpose solutions for semi-dynamic variability in some domains (e.g, math libraries [11] or the Linux kernel [12]) that adapt at load or initialization time to the actually available hardware features, these solutions are limited in scope, require manual intervention, and typically involve costly extra indirections via proxies [13] or inherently fragile means of fine-grained run-time code patching [12]; often they have to prevent inlining.

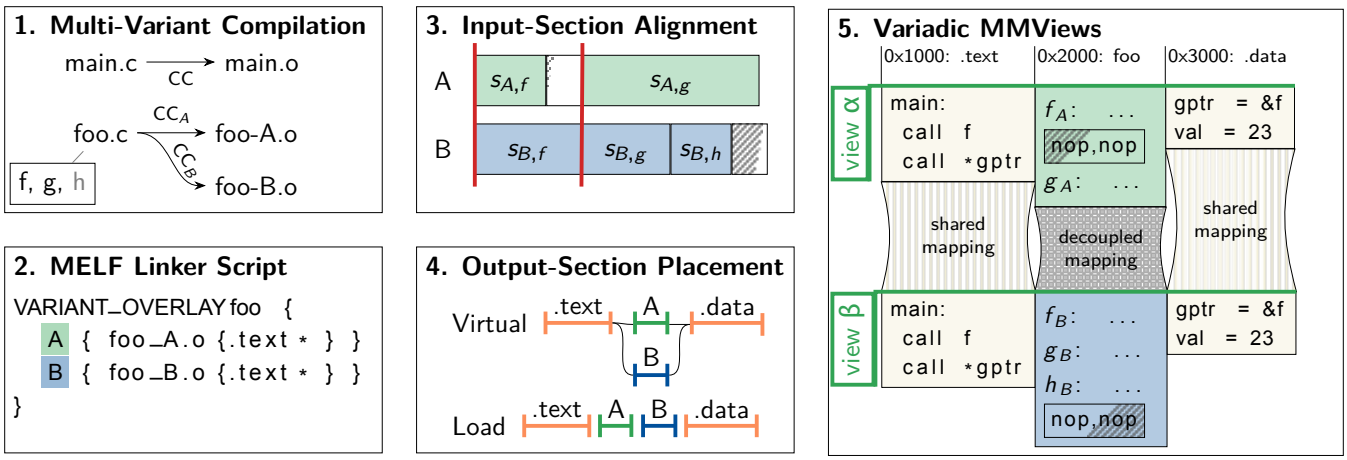


Figure 1: Overview about the MELF approach. At compile time (1), parts of the program are compiled into multiple variants (A and B), which are captured and organized in the linker script in the variant overlay `foo` (2). In the linker, the matched input sections are aligned (3) and the resulting output sections (4) are placed in the virtual- and load-address space. At run time (5), the variants are loaded into different MMViews, which share everything but the decoupled regions; common symbols and pointers are stable across views.

About this Paper

We present *multivariant ELF (MELF)* as an easy-to-apply approach for the inclusion of multiple compile-time variants within the same binary and flexible switching between them at run time on function/section granularity. MELFs are implemented solely on binary level, hence mostly independent of the employed languages and compilers (as long as they produce ELF-compatible objects), which also makes them easy to apply to existing software projects. Function variants are aligned by the MELF linker to the same virtual address, so that existing pointers or relocations remain valid even in case of a variant switch at run time. They can optionally be loaded by the MELF loader into additional in-process address spaces (with the MMViews kernel extension, taken from [14]), where multiple variants can coexist at the same time to be applied on a per-thread level.

In particular, we claim the following contributions:

- We provide the MELF concept, as an end-to-end solution for semi-dynamic variability.
- We describe our MELF linker (an extension to LLVM's LLD linker) and the MELF loader.
- We demonstrate the MELF benefits and costs in four case studies from different domains.

2 The MELF Approach

The MELF approach (see Fig. 1) provides semi-dynamic variability on function granularity for compiled functions by aligning functions (and data) with a modified LLD at link time. At run time, either one of variants is loaded into the process's address space or, with the help of the MMViews, multiple variants can coexist simultaneously. We describe our approach for *executable and linking file format (ELF)* and Linux processes, but are confident that our approach is generalizable to other binary formats (e.g., COFF, Mach-O) and process models.

2.1 System Model

We target programs written in compiled languages (e.g., C, C++, Rust, ...) that organize their binary code in regular, hierarchically-called functions (e.g., unlike Haskell, Forth). For a subset of all functions (i.e., not `main()`), it is intended to include multiple function variants in the final binary. For these functions, the signature (including the mangled symbol name) must be equal and their side effects on the program's object space must be compatible. The compiler must be able to put functions into their individual binary section.

2.2 Compile-Time: Static Variant Generation

First (Fig. 1, step 1), we have to statically generate multiple variants of our functions at compile time. These static variants can, for example, originate from translating the same translation unit multiple times with different compiler flags or CPP configurations. But also, manual variant encoding (e.g., directly in assembler) or programmatically in the compiler via guided-function specialization [15] is possible.

In Lst. 1, we show that this variant generation is simple with modern build systems, like CMake [16]. In the example, which is taken from our SQLite3 case study (Sec. 3.1), we compile the SQLite source files twice into two static

libraries.[1] Both libraries are compiled with different predefined CPP macros and linked into the `main` executable.

```
# Collect SQLite 3 Source files
file(GLOB SRCS sqlite3/*.c)

# The Non-Debug Version
add_library(sql-ndebug STATIC ${SRC})
target_compile_options(sql-ndebug -DNDEBUG=1)

# The Debug Version
add_library(sql-debug STATIC ${SRC})
target_compile_options(sql-debug -DSQLITE_DEBUG=1)

# Case-Study Executable: main
add_executable(main main.cc melf_loader.c)
target_link_libraries(main sql-ndebug sql-debug ..)
```

Listing 1: Multi-Variant compilation with CMake

Besides the multi-variant compilation, the ELF [17] standard forces us to put each function and each data object into their own section. In order to understand this requirement, we now take a quick detour into the ELF standard, which is also necessary to understand the MELF linker.

Excursus: ELF Sections vs. Functions The *executable and linking file format (ELF)* is a format that is used for linking *and* for loading programs. An ELF contains multiple byte streams (code, data, debug info, ...) that are arranged in two views: In the link view, those byte streams are called *sections*, while they are called *program headers* (or segments) in the load view. Additionally, the link view makes use of *symbols* as named offsets into a section. Further, *relocations* specify how to edit a byte stream while linking it to a virtual address. In a nutshell, the linker arranges the link-time sections into load-time segments while resolving (most) relocations.

The basic unit of linking is the section and not a (language-level) function or (global) data object. In fact, the linker has no idea about those, and it cannot (due to compile-time resolved relocations) break up a section back into smaller pieces. Therefore, as we want to align the function's start address, we have to instruct the compiler[2] to put each function (and data-object) into its own section, named like the (mangled) function name. For example, the C++ function `void foo(int)` will end up in the section `.text._Z3fooi`.

For data objects (i.e., global variables and read-only data), we require that all variants share the same (data-) object space. For this, the linker has to throw away all but one instance, which requires each variable to be located in its own section[3]. Furthermore, we restrict the program's interpretation of the shared data objects: As objects can be accessed from different variants, we *require* that the interpretation of objects, statically *and* heap allocated, must be compatible across all variants. This means that matching `struct` fields have to be aligned, that language-level types have to be of equal size, and that variants must have a common understanding of the object state. Nevertheless, this restriction holds for many automatically-applied program transformations as they are nonfunctional by design. In Sec. 4, we will discuss this topic further.

After the multi-variant compilation, we end up with a set of ELF object files whose sections s can be categorized as follows: A *variant* $v = \{s_{v,1} \dots s_{v,n}\}$ is a collection of sections that should be visible together; all sections of one variant have to have the same *section type* (e.g., code, read-only data, ...), which becomes the *variant type*. A *variant overlay* (overlay, o) is a collection of $|o|$ equally-typed *variants* and a *variant-overlay group* (group) is a set of overlays that are semantically connected. For a program, multiple independent overlays and groups can exist. For example, besides a math-code overlay (non-AVX vs. AVX2), there can another overlay group (code and read-only data) for the SQLite library that allows to en/disable executable assertions. We call all sections that are not covered by a variant the *remaining* sections.

2.3 Link-Time: Virtual-Address Alignment

After the compile-time preparation, the linker generates the multi-variant ELF (Fig. 1, steps 2-4), for which it must match and align sections from the different variants within an overlay such that they end up with the same virtual address. For our implementation, we modified `LLD` [18], the linker of the LLVM project that is a drop-in replacement for the GNU `ld` and `gold`. We will describe the required linker modifications on base of this implementation. However, they should be easily generalizable to other linkers as well.

First (Fig. 1, step 2), the developer must be able to express the relationship between sections, variants, and overlays. For this, we add a `VARIANT_OVERLAY` statement to the linker-script language, which is the command language of `ld` (and `gold`/`lld`). The statement contains multiple variant statements, which define, similar to other commands, patterns that are matched against the *input sections*, which the linker extracts from the object files. The lexically first variant of an overlay is its *primary* variant. In the overview example, we see a linker-script fragment that defines an overlay `foo` with two variants (`A`, `B`), which collect the code (text) sections from the respective `foo_{A,B}.o`.

Thereafter (Fig. 1, step 3), we align the input sections within an overlay: For this, we first match sections from the different variants and identify those sections that occur only in one variant: Starting with all sections captured by the overlay, we group the sections by the key (variant, section name) and select a single section as representative for that key. While there is usually only one candidate section, ELF

[1] Static libraries are fundamentally different from dynamic libraries. They are only collections of object files, (nearly) transparent for the linking process, and induce no run-time overhead.

[2] GCC/Clang: `-ffunction-sections`, MSVC: `/Gy`

[3] GCC/Clang: `-fdata-sections`, MSVC: `/Gw`

$v \backslash n$	f	g	h	X	J
A	$s_{A,f}$	$s_{A,g}$		$s_{A,X}$	$s_{A,J}$
B	$s_{B,f}$	$s_{B,g}$	$s_{A,h}$	$s_{B,X}$	$s_{B,J}$
C	$s_{C,f}$	$s_{C,g}$			$s_{C,J}$

Figure 2: Input-Section Table (extended running example)

section groups[4] and weakly-defined functions[5] can result in multiple candidates. In the former case, we can choose any section as the group representative, in the latter case we apply the usual override semantic for weakly-defined functions, but only within the group. With the representatives, we end up with one section $s_{v,n}$ per (variant, name)–pair and form the overlay's *input-section table*, which tabularizes the results. In the table (Fig. 2) for the (extended) running example, we see that f and g are present in all variants and h is private to variant B. For partially-filled columns (e.g., X) that contain more than one entry, we report an error.

With the table in place, we validate and manipulate the symbol table. With multivariant compilation, the linker will encounter the same symbol, which is a named pointer into a section, multiple times. Instead of reporting an error, we collect duplicate symbols and delete all but the symbol that points to the primary variant. In this process, we verify that each variant's symbol points to the same column and has the same offset.

While we usually report an error if this check fails, some compiler optimizations (e.g., function-body deduplication) can result in two aliased symbols pointing to the same section in one variant but not in the other. However, as we cannot align this section to two different sections in the other variant, we have to solve this rare situation differently: We equip each variant with a jump table (e.g., $s_{A,J}$ in Fig. 2) and insert a `jmp` instruction for one of the aliased symbols. In each variant's jump table, the instruction jumps to the correct section and offset, while we globally redirect the original symbol to the jump-table entry in the primary variant.

With all symbols being aligned within their column, we align the columns by padding each $s_{v,n}$ to $\max_{v_i \in o} s_{v_i,n}$. As this can induce large padding gaps, we use variant-local sections as *gap-filler sections*. Currently, we perform the filling greedy as we did not encounter a situation where a (more) optimal algorithm would be required. In the example (Fig. 1, step 3), the gray areas are padding and $s_{B,h}$ is used for filling the gap in B that $s_{A,g}$ provokes.

After column alignment, we place the variants in the ELF's virtual address space (Fig. 1, step 4). For this, we utilize the fact that the load address (where the loader will copy the section to) and the virtual address (where the section "thinks" it is) can disagree. We combine all sections for a variant (table row) in an *output section*; an established linker-internal concept that acts as an intermediate step between input sections and segments. Each output section is linked (i.e., relocated) to the virtual address of the primary variant, while we load them, by default, sequentially. Thereby, load and virtual address only match for the primary variant. All remaining sections can be linked as usual.

The result of MELF is a regular ELF binary, which is only special with regard for the non-primary output sections, whose load and virtual addresses do not match.

2.4 Run-Time: Multivariant Loading

With the MELF binary constructed, it is time to bring our multi-variant program to execution. As this depends on the indented usage scenario, this section will only provide the necessary primitives from which different use cases can be constructed (see Sec. 3).

First, we have to decide which variant(s) will execute and initialize the program state. As primary variants are loaded to their virtual address, the program automatically starts executing in those variants, and it also loads their data-segment contents. This also requires us to only run the constructors for global variables of the primary variant, which is done by discarding the initialization-array entries of the other variants.

For the usage of MELF's, we provide two operation modes by the MELF run-time loader library: With the *base mode*, only one variant per overlay is active at the same time, which the developer can replace with an explicit call into the run-time library. For this, we use the `mprotect()` system call to make the respective overlay region writable and copy the contents of the desired variant to the primary virtual address. To make overlay and variant regions known to the MELF loader, the linker places symbols with virtual and the load addresses before and after each output section; with these, the program can reference all variants in the program. Usually, the developer will only replace text and other read-only sections, as switching data variants would reinitialize the global variables. Also, for this mode to work, we demand that no thread currently executes or has a call frame for a function from the replaced overlay. This program state is called *global quiescence* [14].

As the base mode is of limited use for multithreaded programs, we also provide the *MMView mode*, based on MMViews [14]. Thereby, multiple variants can be active simultaneously and threads can switch *their* variants for which they only have to be *locally quiescent* (i.e., they do not execute a replaced function). As this mode requires the MMView kernel extension, we give a brief overview of its semantic.

Excursus: MMViews With MMViews, a process can have *multiple*, closely-synchronized, concurrently-active address spaces, which have the same structure: all mappings are equally placed and address-space modifications work si-

[4] e.g., used for deduplicating functions from C++ template expansions.
[5] Weak functions are only used if no non-weak counterpart is defined

multaneously on all MMViews. Also, the contents of most mappings are synchronized by sharing the physical page frames. Only for mappings that the user explicitly marked as *decoupled*, the kernel will establish a copy-on-write mapping, whereby those mappings contain MMView-local memory. Also, threads can switch between MMViews and can create a new MMView by cloning their current view.

The existing [14] MMView Linux extension implements MMViews as separate page-table trees. So, since an MMView is technically a separate address space, they induce higher memory overhead (for the page tables) and increase the TLB pressure if two MMViews are active on the same core. Also, page-table modifications, although the extension synchronizes them lazily, have a higher run-time overhead. However, switching views is rather cheap as the kernel only exchanges a single CPU register.

Coming back to MELF, we use the described extension to execute multiple variants in one process concurrently: We decouple all primary-variant regions, allowing the user to create one MMView for each desired variant combination. With the described base-mode primitives, the user can load different variants into the MMViews. Thereby, an MMView can combine these variants from the variant-overlay groups.

In Fig. 1, step 5, we see two MMViews α and β, which currently have loaded variant A resp. B. We see that the non-multivariant text and data remain shared and only in the overlay region (0x2000-0x3000) is decoupled. Since MMViews have a synchronized structure and the MELF linker aligned the start address of the multivariant functions, all common symbols (e.g., `call f`) and function pointers (e.g., `gptr`) are globally valid and threads in different views can easily co-operate.

With MMViews in place, a thread can switch variants on function-call granularity, for which call and return edges have to perform inverse MMView switches. For this, the run-time library provides a trampoline function (Lst. 2) that switches to the desired MMView, forwards arguments, restores the previous MMView and returns the return value. As the trampoline is not part of an overlay, it can also transfer the control flow between two multi-variant functions. The call protocol for MELFs defines the following call-chain:

1. Call `call_with_helper` with a variant index, function pointer and its arguments
2. Switch to the variant, save old return pointer and replace it by `call_return`
3. Jump to provided function pointer
4. Return from provided function pointer (now returns to `call_return`)
5. Switch back to the variant before the call-chain started
6. Jump to saved, original return pointer, ending the call-chain

```
_threadlocal variant_id_previous = 0;
_threadlocal variant_return = nullptr;
variant_id = 1;
func_pointer = &do_work;
func_arg = 10;
// 1. Call trampoline.
call_with_helper(variant_id, func_pointer, func_arg){
    asm {
        // 2. Switch view
        push variant_id
        syscall_variant_switch
        // Syscall result is old variant id. Save it.
        xchg %rax, variant_id_previous@threadlocal
        // Load new return pointer "call_return".
        leaq call_return(%rip), %r10
        // Exchange return pointer with "call_return".
        xchgq    %r10, (%rsp)
        // Save old return pointer.
        xchgq   %r10, variant_return@threadlocal
        // 3. Jump to function pointer.
        jmp func_pointer
    }
}

// 4. "func_pointer" will return to this function.
call_return(){
    asm {
        // 5. Load old variant id and switch back.
        mov variant_id_previous@threadlocal, %rax
        push %rax
        syscall_variant_switch
        // 6. Return to original return pointer.
        jmp    variant_return@threadlocal
    }
}
```

Listing 2: Trampoline function `call_with_helper` ensures to call `call_return` at the end of the call chain to switch back to the caller's original application variant.

Since the protocol always requires jumping back to the original application variant, we only demand local quiescence per thread.

3 Case Studies

As the MELF approach is a general semi-dynamic–variability method for compiled languages, we now provide multiple case studies to demonstrate the potential of our approach. We will justify, for each case study, its relevance, describe the usage of MELF, and show its benefits with a quantitative evaluation. Thereby, we will only focus on the MMView mode as we consider it the more interesting application mode for MELF.

Benchmark Setup

We cover the server-centric scenarios with a dual-socket system (Intel Xeon Gold 6252, 2.10GHz, 2×24 physical cores, 2 NUMA nodes, 384 GiB DRAM, hyperthreading disabled). Additionally, we use a smaller machine with more restricted hardware (Intel i5-6400, 4 cores, 32 GiB DRAM, no hyperthreading). On the software side, we used Debian

SQLite 3.39.4 (a29f994989)	*(both views)*
`-O3`	Required for scalability
`DEFAULT_MEMSTATUS=0`	Required for scalability
`PAGE_CACHE_OVERFLOW_STATS=0`	Required for scalability
`ENABLE_RTREE=1`	Required for workload
Unify `struct sqlite3_mutex`	Required for MELF compatibility
Perf. View `NDEBUG=1`	**Debug View** `SQLITE_DEBUG=1`
Functions: 1 432	Functions: 1 726
.text=1 008.5 K .rodata=27.4 K	.text=1 294 K .rodata=51.6 K
.data=2.6 K .bss=1.2 K	.data=2.6 K .bss=2.3 K
MELF Overlay	
Aligned Functions: 1 310	Padding: 372.3 K (13.48 %)
VM Size: .text=1 314.3 K .rodata=61.8 K .data=2.7 K .bss=2.5 K	

Table 1: Overview over the SQLite case-study binary

GNU/Linux 11 with an MMView-enabled Linux 5.15 kernel with Spectre and Meltdown mitigations enabled.

3.1 Case-Study: SQLite Asserts

With this case study, we demonstrate that MELF is able to overlay multiple handwritten code variants and that we can performance-isolate both variants for thread-contextualized execution (with MMViews). More concretely, we build a MELF binary that contains two variants of the SQLite library: (1) the *debug view*, where `assert()` statements and additional sanity checks are enabled, and (2) the *performance view*, where these are disabled. Within the same process, multiple threads execute read-only SQL queries, either with the debug view or the performance view. We vary the total number of threads and the number of threads in the debug view, as well as the benchmark machine.

Scenario Justification Unlike compiler-based security measures, executable asserts [7], [8] are inserted manually by the developers to test high-level invariants at run time. They are an intrinsic part of debug builds, which often include extended data structures and code paths to check application behavior. Thereby, assertions not only assist the development of safer programs [9], but they are also an active security measure [10]. However, due to their complexity, size and performance impact, they are usually disabled in production in favor of a performance/release build. With MELF, we can provide a more restricted debug view with enabled assertions, for example, for SQL queries that handle user input. Technically, this case-study is of interest as it shows how to manage multiple variants that interpret data differently.

Workload We use a geospatial proximity search, since handling two-dimensional data requires complex algorithms and data structures. On the list of 2856 UK postcodes, we issue SQL queries that find the geographically closest code that is not further away than 25 km for randomly chosen coordinates in the UK. For handling coordinates, we use SQLite's R-Tree plugin.

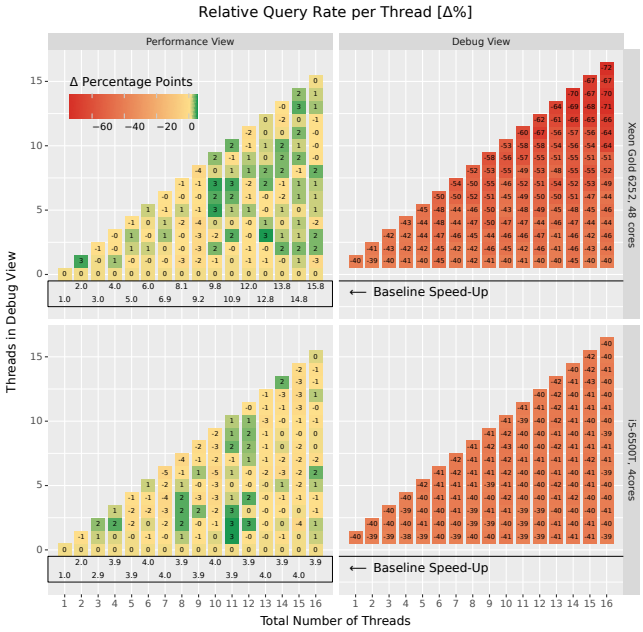


Figure 3: SQLite Performance Measurements

Benchmark In Tab. 1, we provide a comprehensive overview about the used benchmark binary. Since SQLite's default configuration did not scale beyond a single core, we had to disable some statistic features to limit contention. While running both views concurrently worked out-of-the-box for most parts, we had to unify the `struct sqlite_mutex` as the debug view requires additional fields to track mutex ownership. Without a unified data type, the address calculation for array elements differed and provoked a crash. In total, we had to change 30 lines of code.

For a seamless interoperability, MELF aligns 1 310 functions by inserting a total of 372.3 KiB of padding, which is 13.48 percent of the combined size in the virtual address space. MELF already optimized the required padding by using 109/202 view-private functions in the performance/debug view as gap fillers. For the mutable global data in (.data, .bss), we align both variants but only use the debug view's data.

Performance Isolation In order show that the MELF approach is able to isolate the impact of the debug view on threads in the performance view, we run the benchmark with 1 to 16 threads, whereby 0 to 16 threads execute permanently in the debug view, while the others execute in the performance view. We also execute the benchmarks on our 4-core and on our 48-core machine in order to determine if core contention has a significant impact. We execute each benchmark for 60 seconds, record the number of completed SQL queries

In Fig. 3, we show the per-thread SQL-query rate and normalize it to the results where all threads execute in the

performance view (y-axis = 0), which we consider the baseline for this experiment. For the baseline case, we also show the speedup to confirm that contention within SQLite itself is not the cause of performance degradation but only the usage of MMViews and MELF. As expected, we see near perfect speedup on the 48-core machine, while the speedup caps around 4 on the 4-core machine. Please note the highly asymmetric color scale in this figure.

In the debug view, we see a significant impact of the additional assertions and sanity checks on the query rate. As the slowdown on the 48-core machine (-39 % to -72 %) is significantly worse for more threads in the debug view than on the 4-core machine (-38 % to -43 %), we conclude that the additional sanity checks provoke more contention due to additional state locking.

In the performance view, we see that the number of threads in the debug view has no consistent impact on the other threads, and some results even indicate better indicate a higher performance with using MMViews. Therefore, we take a look at the relative standard deviations for the baseline case to determine if these results stem from SQLite itself. While we cannot derive any conclusions from the relative standard deviation for the 4-core machine (0.3 %–12.4 %), the 48-core machine (rel. stdev.: 0.3 %–1.7 %) suggests that the MELF approach also has a small impact on the performance view. If compared to the observed relative query rates (-4 % to 3.4 %), we conclude that MELF has a negative performance impact of around 1 percent and adds around 2 percent of jitter. Nevertheless, in relation to the impact of globally-enabled assertions, the MELF approach isolates the impact of additional sanity checks in SQLite successfully.

3.2 Case-Study: Thread Pools on Heterogeneous Instruction-Set Machines

With this case study, we show that MELF eases the programming of non-homogeneous multicore machines where cores share a common subset *instruction-set architecture (ISA)* but have additional heterogeneous ISA extensions. More concretely, we provide a thread-pool abstraction (see Fig. 4) that accepts jobs together with a hint on which core type the job will run best. Depending on the current load, the pool schedules the job (preferably) on a hinted core where it uses a MELF-prepared code view that exploits the core-specific ISA extensions or on another core with a different code view that is optimized for that core. Thereby, the thread-pool user fully utilizes her heterogeneous architecture without the need for adapting her code paths for the specific architecture.

Scenario Justification While the first non-uniform multicores (e.g., , ARM big.LITTLE [19]) came with a unified ISA, recent work [20]–[22] investigates on the performance and energy benefits of heterogeneous ISAs. However, ISA diversity poses a programmability challenge as programmer are not keen to distribute their program/data

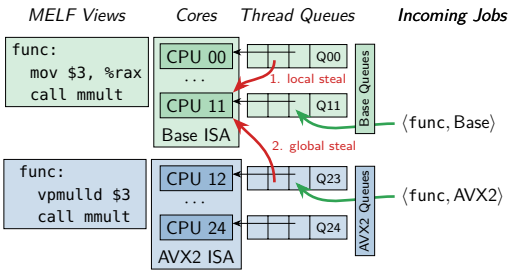


Figure 4: Heterogeneous-ISA Thread Pool

flow manually over different ISAs. Therefore, researchers proposed fault-and-migrate [23], cross-core invocation [24], and multi-kernel [25] methods to manage this variability. With MELF, we take a step towards the seamless integration of heterogeneous ISAs into our programs. Technically, this case-study is of interest as we make use of different cross-cutting compiler options on instruction level, something that is not easily expressible on a language or ifdef level.

System Model As we have no heterogeneous-ISA machine at hand, we simulate one by virtually dividing one NUMA node of our 48-core machine into two partitions (see Fig. 4): On the 12 AVX2 cores, modern AVX/AVX2 vector instructions are available, while the other 12 cores lack this ISA extension.

Work Load For our benchmark, we choose two job types that benefit differently from the AVX2 instructions: While jobs with a recursive Fibonacci (n=36, Base/AVX: 57.9 ms) do not benefit at all, the duration of a Matrix-multiplication (565×565) job drops from 58.3 ms to 38 ms on the AVX2 core. For the matrix multiplication, we use the Eigen C++ library (v3.4), which uses explicit ISA specialization according to the given compiler flags. Please note, that we have chosen the parameters such that the base-core execution time match. As work load, we submit 1000 jobs with 0 to 100 percent of the jobs being matrix multiplications (see Fig. 5) and record the end-to-end latency of those 1000 jobs as well as the accumulated job execution time.

Thread-Pool Variants Based on Eigen's non-blocking thread pool, which already implements thread-local queues and work stealing, we build three thread-pool abstractions that all take a function pointer and a scheduling hint as a job description: The **1 Pool, Base only** variant executes all jobs on a single 24-worker thread pool and only uses code without AVX2 instructions; the scheduling hint is ignored. The **2 Pool** variant uses two 12-worker thread pools, one for the base cores and one for the AVX2 cores; each core executes code specialized for its ISA and workers are pinned to its core; *no* stealing happens between the pools; and the scheduling hint selects the thread pool. The **1 Pool, MELF** variant uses a single 24-worker pool that utilizes MELF: Each worker thread is pinned to its core and executes in a MELF code view that is specialized for its ISA. If a worker

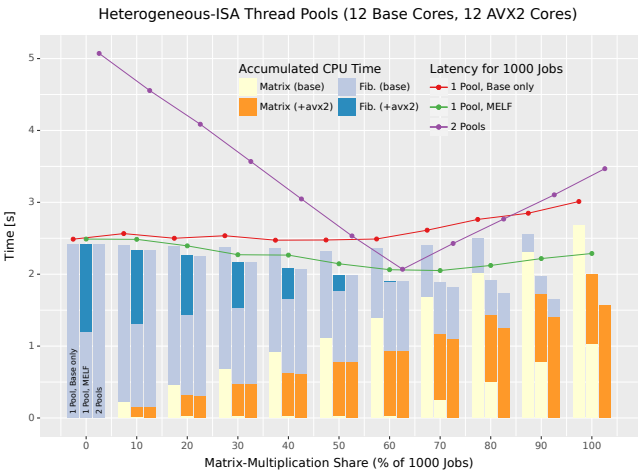


Figure 5: Latency and accumulated processing time for a mixed work load on a 24-core heterogeneous-ISA machine with different thread-pool configurations. To scale latency and processing time, the processing time was divided by 24.

queue runs empty, the worker first tries to steal from workers with the same ISA (see Fig. 4, 1. local steal) before stealing from other ISAs (2. global steal). Please note that stealing from a foreign ISA queue works seamlessly as the local MMView exposes the same functions but implemented with different instructions.

Results In Fig. 5, we show the evaluation results on one NUMA node of our 48-core machine, where we compare the three pool variants with respect to required processing time. Please note, that we divided the accumulated processing time by 24 to match its scale to the end-to-end latency. The remaining difference between latency and processing time stems from pool overheads and execution phases where not all workers execute jobs.

For the Base only variant, processing time is, as expected, only spent in the base code view (less bright colors). The slight increase in both curves stems from the increased cache pressure if 24 cores execute matrix multiplications in parallel compared to executing recursive Fibonacci calls on the (cached) stack. Although the 2 Pool variant spends the least amount of processing time, its end-to-end latency for 1000 jobs is significant at both ends as it only utilizes 12 cores at 0 and 100 percent matrix-multiplication jobs. Also, it achieves its best latency at 60 percent multiplications, which is expected from the ratio between a Fibonacci job (57.9 ms) and an AVX2-Matrix multiplication (38 ms).

Finally, the MELF-enhanced 1 Pool variant, performs better in both dimensions: Compared to Base only, it uses less processing time as it actually utilizes the AVX2 instructions, whereby also its latency is better. Compared to the 2 Pool variant, it always utilizes all cores resulting in a consistently low latency and only when more than 60 percent of the submitted jobs are Matrix-multiplications requires more processing time.

3.3 Case-Study: Profiling in memcached

In this case study, we dynamically en-/disable compiler-introduced function-level profiling on a per-thread basis with MELF and MMViews in `memcached` (v1.6.10). Similar to the SQLite study, we combine two memcached variants in one binary:(1) In the *profiling view*, the compiler (with the `-pg` flag) introduced `mcount()` calls into function prologues that record the invocation, while (2) the *performance view* contains the same functions but without profiling code. On a per-connection basis, worker threads either select the profiling or the performance view. For our benchmark, we gradually change the number of profiled connections and measure the request handling time within `memcached`.

Scenario Justification As developers cannot emulate complex production environments, it is often up to the DevOps team to detect and explain performance anomalies after deployment. For this function-level profiling, as provided by `gprof` [26], would provide precise insights about call frequencies and caller–callee relationships, but its cost prohibits us to have it permanently enabled. Also, in a multi-tiered environment, where only some clients incur a certain anomaly, it is desirable to enable profiling only selectively for certain threads and requests. Because `gprof` consists of both, a compiler instrumentation to modify function translation and a statically-linked profiling library, developers are unable to define exclusive code paths they want to profile. They can either profile the whole application or nothing at all. Thanks to MELF, the DevOps team can enable `gprof`-profiling dynamically and selectively, thus limiting the impact to a minimum.

Technically, this case-study is of interest as we reduce contention on cross-cutting features.

Work Load As a work load for our multi-variant `memcached` server, we use the `memtier` benchmark, which is a specialized benchmark for key-value databases [27]. On the client side, we use its default SET–GET ratio of 1:10 and start 16 threads with 50 clients each, resulting in 800 clients with 800 active connections to *memcached*. We execute the benchmark on the same machine, but pin `memcached` to one NUMA node, while pinning `memtier` onto the other. We record data request latencies until each view has serviced 100 million requests.

Benchmark Unlike other servers, `memcached` has an event-based design and the worker threads execute a state machine for each connection. Thereby, connections can be easily rebalanced between workers and one worker routinely handles many connections. To match the 16 `memtier` threads, we start `memcached` with 16 worker threads that, together, service all client requests.

For new connections, we decide whether it will execute in the performance or the profiling view, mimicking scenar-

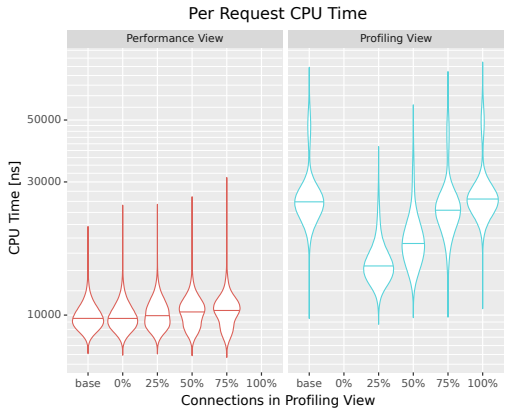


Figure 6: Memcached Performance Measurements for 1M sampled Data Points per Violin

ios where certain IP ranges or customers are profiled. In our benchmark, we use enable profiling for the first N (0 %-100 %) connections to demonstrate the impact of profiling. As long as the profiling percentage threshold is not met, each connection will be profiled. Because `memcached` distributes connections round-robin across workers, each worker thread will serve both, profiling and performance connections. Consequently, every worker thread switches between performance and profiling view continuously across the whole benchmark, which increases TLB pressure since they live in distinct address spaces. Nevertheless, as we will show, this penalty is still better than to enable profiling globally and could even be improved by a profiling-aware connection scheduling.

For our measurement, we record the execution time of the `drive_machine()` function, which is responsible for executing the per-connection state machine, making the function crucial for the request latency. For each benchmark run, we record request latencies until we reach 100M data points for each view. Therefore, for the mixed-mode benchmarks (25 %-75%), we will end up with 200M data points.

Results In Fig. 6, we show violin plots for the `drive_machine()` execution times separated by requests serviced that were serviced in the profiling or the performance view. Please note, that the 25% performance violin on the left matches the 25% profiling violin on the right as both stem from the same benchmark run. Also, we include the *base* variants, which show the results for a `memcached` server with statically enabled or disabled profiling without MELF or MMViews. For preparing the violin plots, we limit each data set (100M data points) to the [0.01%, 99.99%]-interval to remove outliers and randomly sample 1 million representative data points.

By comparing the base variants, we see that profiling has a significant impact on `memcached`'s most important function and increases its execution time by 175 percent. Further, for both views, the violin with the maximum number performance/profiling connections match the results of the base variant (i.e., performance base $\leftrightarrow$ 0% performance). From this, we can conclude that MELF enables us to enable and disable profiling dynamically at run-time without having a continued run-time impact.

For the profiling view, we see that the impact of profiling *per request* increases if more connections use the profiling path. This behavior can be explained by looking at the `gprof`-induced into the code: For each function, `gprof` allocates a counter variable that the compiler-introduced `mcount()` function uses to keep track of the number of invocations. As the activated `memcached` logic is rather small and executed by 16 threads in parallel, all workers in a profiling view access the same small set of per-function counters. Together with the fact that `gprof` does not even cache-align these counters, profiling results in many cross-core cache invalidations. Further, this effect scales with the percentage of threads working in the profiling view as more cache conflicts happen.

For the performance view, we see a slight increase in the median and tail latencies if more connections are shifted to the profiling view. Since workers need to switch views if the currently active MMView does not match the next processes connection, MELF increases the TLB pressure, impacting also workers in the performance view. Also, the frequent cache invalidations in the neighboring profiling view increases the cache-coherence traffic and puts a burden on the memory bandwidth. Nevertheless, even with 75 percent of all connections being profiled, the median over the base variant only increases by 7.6 percent for performance connections, which is far less than activating profiling globally. Inspired by these results, one could restrict profiling to a single worker thread or a small set of connections to get a statistic picture of the whole `memcached` process without inflicting the described cache conflicts from invocation counters.

In total, the MELF approach was able to make the static `gprof` method dynamically and selectively applicable without requiring a restart of the process and without touching the compiler.

3.4 Case-Study: ASAN in MariaDB

With this case study, we demonstrate that MELF is able to handle multiple variants in complex C++ projects. We compile MariaDB (> 20000 functions) once with (`-fsanitize-address`) and once without address sanitizer [5] and use the MELF linker to overlay both variants in the same binary. At run-time, we decide on a per-user basis whether a client's SQL queries are executed with sanitized or unsanitized MMView.

Scenario Justification Sanitizers [28], like AddressSanitizer [5] and UBSan [29], are often implemented as compiler transformations and they are usually used at de-

MariaDB (10.11)			
ASAN View		**Normal View**	
Functions: 21 448		Functions: 20 986	
.text=16 777.4 K		.text=4 661.6 K	
.rodata=1 082.2 K		.rodata=1 079.2 K	
.data=173.9 K		.data=173.9 K	
.bss=218.3 K		.bss=218 K	
MELF Overlay			
Aligned Functions: 20 615		Padding: 12 487 K (33.87 %)	
VM Size: .text=16 934 K	.rodata=1 099 K	.data=180 K	.bss=224 K

Table 2: Overview of the MariaDB case-study binary

Clients	Normal View		ASAN View	
Normal / ASAN	Median	99%	Median	99%
24 / 0	63 us	73 us	–	–
18 / 6	63 us	74 us	90 us	104 us
12 / 12	63 us	74 us	89 us	102 us
6 / 18	64 us	74 us	90 us	102 us
0 / 24	–	–	89 us	102 us
Base w/o MELF	47 us	55 us	89 us	100 us

Table 3: Query Latency for Sysbench `oltp_point_select` benchmark on MariaDB with and without AddressSanitizer (ASAN).

velopment time to find bugs. However, due to their high overheads, they are then disabled in production, although they could provide an additional level of sanity checking for code that handles user input. With MELF, we enable AddressSanitizer, which was found to be the most common sanitizer [28], for individual database users in MariaDB, whereby we mimic a trusted–untrusted customer model. Technically, this case-study is of interest as MariaDB is a multi-threaded, large server application. With ASAN being strongly invasive on the code and data path, it helps to understand how MELF scales for large code bases.

Work Load As a work load, we use the `sysbench` [30] `oltp_point_select` benchmark. On our 48-core machine, we execute and pin MariaDB to NUMA node 1, while sysbench runs on NUMA node 2. We start MariaDB in the one-thread-per-connection mode, and always have 24 concurrent sysbench connections. To satisfy the mentioned trusted–untrusted customer model, we execute two sysbench instances, each of which connects as different database user. To vary the load between the ASAN/no-ASAN view, we vary the distribution of the 24 connections between both instances. We use the output of sysbench, which records the end-to-end latencies per transaction, as our result data.

Results In Tab. 2, we see an overview of the MariaDB binary produced by the MELF linker. First, we see that the application of ASAN increases the number of functions, as the compiler cannot inline and eradicate some of the smaller functions. We also see that the ASAN variant's text section is 2.6 times larger than the normal text section. Together with the fact that 98 percent of the normal view's functions had to be aligned and, therefore, could not be used for gap filling, explains the larger percentage of padding bytes (33.87 %).

In Tab. 3, we show the end-to-end latency results for the `oltp_point_select` benchmark. First of all, we see that AddressSanitizer has a significant impact on the performance of MariaDB as it increases the median latency by 89 percent. This latency penalty is also inflicted on clients whose queries are processed in the ASAN view. However, also clients in the normal view have a 36 percent increased query latency. This increase can be explained by the fact that the ASAN run-time library still has to intercept and wrap heap allocations, which are known to have a major impact on query performance[31], in order to keep its shadow-memory map up to date. However, in the normal view, MELF only removes the additional checks from the query processing and the additional overhead from the run-time library remains.

4 Discussion

In the following, we discuss limitations and benefits of the MELF approach.

Multi-Variant Data As we have discussed in Sec. 2.3, the MELF approach is currently limited to a strict data-object sharing semantic, where all variants share the data sections of the primary variant. For this, the data and its interpretation have to be compatible in all variants, which can, as we have seen with SQLite (Sec. 3.1), require some manual program modifications.

The following program demonstrates this limitation as it is incompatible in three different dimensions: (1) If a lock is allocated in A, the object would be too small to usage in B, (2) the field `L` has different offsets, and (3) both variants have a different idea about the lock state (`1` vs. `-1`).

```
// Variant A                    // Variant B
struct lock { int L; }          struct lock {int 0; int L;}
#define LOCKED 1                #define LOCKED (-1)
```

Supporting these cases in general is impossible, as it would require complete program understanding on the language level. However, for many cases, one could use semi-automated transformer functions [13], [32], [33], known from dynamic software updates, to synchronize two copies of the data.

For data initialization, we use the variant that is active at the initialization time. Therefore, we use the global data segment and invoke all global constructors in the primary variant. Function-local static variables are, in line with C/C++ semantics, initialized at the first call of the respective function and, thus, in the context of the then active variant.

Besides strict data sharing, the MELF linker also supports a strict data-object partitioning. For this, the linker has to keep all data sections, let each variant only reference its own

data sections, and we would run the constructor of all variants at program start. In this use case, the developer has to ensure that objects do not flow (i.e., across the univariant parts of the program) across variant boundaries. This mode could be useful for using multiple incompatible versions of libraries that make heavy use of global state.

MMView Dependency We acknowledge that MELF plays out its benefits particularly in combination with MMViews, which we use to back the same virtual-address range with different contents depending on the active thread of a process. The MMView approach has disadvantages, such as memory overhead and increased TLB pressure [14]. Because the exact runtime overhead of MMViews highly depends on the size of virtual memory and its physical data (plain data in RAM, file-backed mappings), a general overhead cannot be quantified. Furthermore, the measurable effect on the TLB directly correlates to a thread's view switch frequency and memory access patterns, which is individual to every application. For view creation and switching, a mean runtime penalty of 7µs with a standard deviation of 6µs has been measured on earlier benchmarks for memcached and MariaDB [14]. In another recent study of memcached, the cost of MMViews were only measurable for context-switches between different views. In general, however, a transition from one view to another is comparable to a context-switch between two threads of two different processes. As an alternative, multithreaded MELFs could be facilitated through CPU-assisted segmentation [34], such as supported by the IA-32 architecture [35]. With segmentation, we would load every variant into its own segment and each thread could select its variant by setting its code-segment selector register accordingly. On the IA-32 platform, where call and jump instructions implicitly use the code segment, this would be equivalent to MMViews. Without the separate address-space clones, the memory and TLB overheads would be replaced by a minor offset calculation overhead that segmentation entails.

Although segmentation contradicts Linux's flat memory model, MELF binaries could easily be supported if (a) the kernel provides means to initialize and switch hardware segments and (b) if the code-segment register is preserved between thread switches. Unfortunately, segmentation as a virtual memory primitive is currently not in fashion on modern platforms. Particularly, it has been removed from AMD64 [35] and was never available on RISC architectures. Given that segmentation has other advantages, such as safety benefits, we would applaud a renaissance of this virtual memory primitive. However, even without segmentation, we could theoretically implement text variants, using position-independent code coupled with the segmentation remnants in AMD64 (FS/GS register) to facilitate variant-adherence for indirect jumps. However, this would require intrusive linker and compiler modifications.

Switching For both modes (base and MMView), we demand that switching the variant takes only place at function boundaries. This limitation stems from the fact that MELF only aligns function start addresses, but all other intra-function addresses could be unaligned. For example, saved return addresses may not be valid in the other variant. However, with additional compiler support, this quiescence requirement could be weakened: For example, if the compiler would also align call sites and would make the call frames at those call sites compatible across variants, we could switch variants flexibly at every call and return edge. Such an extension could be beneficial for supporting workloads on heterogeneous ISA as it would, for example, ease thread migration between different ISAs without stack rewriting [36].

Applicability and Benefits With our case studies, we have shown that the MELF approach is applicable to a wide range of programs. By covering not only C but also C++ projects, which result in more complex object files (e.g., C++templates are a main user for COMDAT), we have demonstrated that MELF works on multiple programming languages. Further, as our approach only requires a compiler to produce "sectioned" object files, we are in principle language agnostic and widely applicable.

Also, MELF is agnostic to the source of the code modification. As shown, we support automatically-introduced compiler transformations (e.g., profiling) as well as manually-encoded variants (e.g., SQLite). Thereby, MELF covers more scenarios than pure language-based methods, like *aspect-oriented programming (AOP)* [37]. Further, as MELF prepares everything at link time, static binary validation could make MELF safer than dynamic-binary instrumentation (e.g., Intel Pin [38]), which was criticized to ease an exploiter's life [39].

Further, we have shown that MELF's semi-dynamic approach to variability is able to cover a wide range of use cases that are security-related (assertions, ASAN), provide DevOps with deeper insights (profiling), and ease the support of coming hardware generations (heterogeneous ISAs). We believe that especially the DevOps and the heterogeneous-ISA scenario will require semi-dynamic variability, since: (1) We need more dynamically-observed metrics to understand our complex systems down to the individual hot path. (2) Extensible architectures, such as RISC-V platform [40], with its many ISA compatibility levels, will boost the spreading of heterogeneous-ISA machines. (3) In many settings, we simply have not the choice to drop existing applications in favor of from-scratch developed software.

Although three of our four case-studies do not dynamically switch views during runtime, we were able to gain reasonable performance isolation in each application scenario: For profiling in memcached we achieved performance isolation of profiling connections and do dynamically switch a thread's view based on the connection currently being

served. In the other case-studies, we were able to obtain: (a) Performance isoliation and improved robustness for dynamic assertions in SQLite. (b) Performance maximization via ISA-specialized function variants with thread-pools. (c) Performance isolation and improved security for ASAN in MariaDB. Additionally, function pointers work "out-of-the-box" for MELF, which eases a programmer's life.

We also imagine that MELF can be used in an embedded setting, where no MMU is available to implement MMViews: For these machines, MELF can prepare variant overlays of in-flash text segments, which then can be exchanged at run-time by partially rewriting the flash memory. Thereby, multiple software variants can be supported in one device without inducing indirection overhead.

Also, we have seen that MELF's function alignment only induces moderate memory overheads (see Tab. 1), while the run-time overhead in combination with MMViews depends on the concrete case study. Nevertheless, even in the `memcached` case study, where threads switch on a regular basis between views, the run-time overhead was limited to less than 8 percent. Furthermore, Fig. 6 shows that the median runtime latency in the performance view is equal for the base and 0% variant.

Summarized, MELF is a cheap, language-agnostic method to lift static code variability to the semi-dynamic level. MELF is widely applicable and provides us with a framework for further explorations of semi-dynamic variability.

5 Related Work

Technically, text overlays [41], [42] are a closely related topic: They were used to reduce a program's primary-storage requirement by loading only the currently used subset of functions into the memory. While overlays have a renaissance [43], [44] for managing complex memory hierarchies, they are fundamentally different as they partition one program to fit it into a smaller memory. In contrast, MELF overlays multiple but similar programs in one binary and, with MMViews, execute those variants concurrently.

On the language level, aspect-oriented programming [45] if applied dynamically [46], [47] is similar to MELF. However, as aspects only add code before, after, or around function (calls), it does not support variants that stem from generic and cross-cutting code transformations.

Function Multiversioning [48] is a GCC extension to generate multiple versions per function, each of which specialized for the availability of different instruction-set extensions. The loader selects one variant on function granularity, which, unlike with MELFs, cannot be changed later on.

Fat binaries support multiple processor architectures by embedding program versions for the different processor types into one executable or library [49]–[54]. The variant to execute can be either selected directly by the operating system [54] or through a polyglot opcode string that is interpretable by both architectures [52]. Nextstep's *Mach-O* format, which was later adopted by Mac OS X, even supports "multifat" binaries that allow more than two different architectures (68K, x86, HP PA-RISC, SPARC) [50], [53], [54]. Going one step further, Cha et al. propose a system for generating multi-architecture binaries that, in contrast to fat binaries, use the same program string which is transformed in a way to be correctly interpretable by multiple processor types [55]. Similarly to the architecture heterogeneity of fat binaries, the *Windows Portable Executable* format has support for multiple platforms as it contains a DOS and Windows program in parallel [56]. Whereas the DOS part is usually just a small stub nowadays, it has been used to ship binaries that work under DOS and Windows in the past. In contrast to MELF, in all these approaches the variant selection covers the whole program, that means it is determined at process start, and cannot be changed later.

6 Conclusion

Multivariant ELF (MELF) is as a binary-level approach for the inclusion of multiple compile-time variants within the same binary and flexible switching between them at run time on function/section granularity. This facilitates the implementation of *semi-dynamic variability*, that is, dynamic switching between feature-variants at run time that are nevertheless generated and known at compile time, whereby even highly cross-cutting compiler features become configurable at run time. In combination with a kernel extension for in-process address spaces, this even works on the level of individual threads.

MELFs are implemented solely on binary level and mostly independent of the employed languages and compilers. Function variants are aligned by the MELF linker to the same virtual address, so that existing pointers or relocations remain valid even in case of a variant switch at run time. Thereby, MELFs are relatively easy to apply to existing projects. We demonstrated this on the example of four case studies, ranging over a wide range of multithreaded C and C++ projects. In all cases, MELF was able to isolate the costs and benefits of the compiler/developer-induced code variants to those threads, that use it at run time.

Acknowledgments

We thank our anonymous reviewers and our shepherd Kenji Kono for their constructive feedback and the efforts they made to improve this paper. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 468988364, 501887536.

References

[1] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1107–1116. DOI: 10.1109/IPDPSW.2013.207.

[2] C. Cowan, C. Pu, D. Maier, *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (SSYM '98)*, San Antonio, Texas: USENIX Association, 1998, p. 5.

[3] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguardtm: Protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM '03)*, Washington, DC: USENIX Association, 2003, p. 7.

[4] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[6] C. Courbet, "NSan: A floating-point numerical sanitizer," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21)*, Virtual, Republic of Korea: Association for Computing Machinery, 2021, 83–93, ISBN: 9781450383257. DOI: 10.1145/3446804.3446848.

[7] S. Saib, "Executable assertions - an aid to reliable software," in *1977 11th Asilomar Conference on Circuits, Systems and Computers, 1977. Conference Record.*, 1977, pp. 277–281. DOI: 10.1109/ACSSC.1977.748932.

[8] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992. DOI: 10.1109/2.161279.

[9] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray, "Assert use in github projects," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 755–766. DOI: 10.1109/ICSE.2015.88.

[10] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 866–879. DOI: 10.1109/SP.2015.58.

[11] *Intel® oneapi math kernel library*, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html (visited on 12/23/2022).

[12] J. Corbet, *Smp alternatives*, Dec. 2005. [Online]. Available: https://lwn.net/Articles/164121/ (visited on 12/22/2022).

[13] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, Ottawa, Ontario, Canada: ACM, 2006, pp. 72–83, ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1133991.

[14] F. Rommel, C. Dietrich, D. Friesel, M. Köppen, C. Borchert, M. Müller, O. Spinczyk, and D. Lohmann, "From global to local quiescence: Wait-free code patching of multi-threaded processes," in *14th Symposium on Operating System Design and Implementation (OSDI '20)*, Nov. 2020, pp. 651–666.

[15] F. Rommel, C. Dietrich, M. Rodin, and D. Lohmann, "Multiverse: Compiler-assisted management of dynamic variability in low-level system software," in *Fourteenth EuroSys Conference 2019 (EuroSys '19)*, (Dresden, Germany), New York, NY, USA: ACM Press, 2019, ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303959.

[16] *CMake – Cross platform make*, http://www.cmake.org/, visited 2023-01-03. [Online]. Available: http://www.cmake.org/.

[17] *Elf(5) - format of exectuable and linking format (ELF) files*, Linux Progammer's Manual, Mar. 2021. [Online]. Available: https://man7.org/linux/man-pages/man5/elf.5.html.

[18] *LLD - the LLVM linker*. [Online]. Available: https://lld.llvm.org/ (visited on 01/03/2023).

[19] P. Greenhalgh, "Big. little processing with arm cortex-a15 & cortex-a7: Improving energy efficiency in high-performance mobile platforms," *white paper, ARM Ltd*, 2011.

[20] S. K. Bhat, A. Saya, H. K. Rawat, A. Barbalace, and B. Ravindran, "Harnessing energy efficiency of heterogeneous-isa platforms," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 65–69, 2016.

[21] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '21, Virtual Event, Hong Kong: Association

for Computing Machinery, 2021, 187–199, ISBN: 9781450382878. DOI: 10.1145/3433210.3453112.

[22] W. Lee, D. Sunwoo, C. D. Emmons, A. Gerstlauer, and L. K. John, "Exploring heterogeneous-isa core architectures for high-performance and energy-efficient mobile socs," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI '17, Banff, Alberta, Canada: Association for Computing Machinery, 2017, 419–422, ISBN: 9781450349727. DOI: 10.1145/3060403.3060408.

[23] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416660.

[24] S. Cho, H. Chen, S. Madaminov, M. Ferdman, and P. Milder, "Flick: Fast and lightweight isa-crossing call for heterogeneous-isa environments," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 187–198. DOI: 10.1109/ISCA45697.2020.00026.

[25] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the programmability gap in heterogeneous-isa platforms," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, Bordeaux, France: Association for Computing Machinery, 2015, ISBN: 9781450332385. DOI: 10.1145/2741948.2741962.

[26] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, 120–126, 1982, ISSN: 0362-1340. DOI: 10.1145/872726.806987.

[27] RedisLabs, *Memtier benchmark on github*, https://github.com/RedisLabs/memtier_benchmark, visited: 2023-01-02.

[28] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010.

[29] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in llvm," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 633–647, 2017.

[30] A. Kopytov, *Sysbench – scriptable database and system performance benchmark*. [Online]. Available: https://github.com/akopytov/sysbench.

[31] D. Durner, V. Leis, and T. Neumann, "Experimental study of memory allocation for high-performance query processing.," in *International Conference on Very Large Databases (VLDB)*, 2019, pp. 1–9.

[32] I. Lee, "Dymos: A dynamic modification system," Ph.D. dissertation, University of Wisconsin-Madison, 1983. [Online]. Available: www.cis.upenn.edu/~lee/mydissertation.doc.

[33] M. Hicks, J. T. Moore, and S. Nettles, "Dynamic software updating," *SIGPLAN Not.*, vol. 36, no. 5, 13–23, May 2001, ISSN: 0362-1340. DOI: 10.1145/381694.378798.

[34] J. B. Dennis, "Segmentation and the design of multiprogrammed computer systems," *Journal of the ACM*, vol. 12, no. 4, pp. 589–602, 1965, ISSN: 0004-5411. DOI: 10.1145/321296.321310.

[35] *Intel® 64 and ia-32 architectures software developer's manual, combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4*, Intel Corporation, 2022. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671200.

[36] K. Makris and R. A. Bazzi, "Immediate multithreaded dynamic software updates using stack reconstruction," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX '09, San Diego, California: USENIX Association, 2009, pp. 31–31.

[37] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Communications of the ACM*, pp. 59–65, Oct. 2001.

[38] *Pin - a dynamic binary instrumentation tool*, Intel Corporation, Santa Clara, California, USA, 2022. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/98650/Pin/doc/html/index.html.

[39] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel, "Pwin – pwning intel pin: Why dbi is unsuitable for security applications," in *Computer Security*, J. Lopez, J. Zhou, and M. Soriano, Eds., Cham: Springer International Publishing, 2018, pp. 363–382, ISBN: 978-3-319-99073-6.

[40] A. Waterman and K. Asanović, Eds., *The risc-v instruction set manual, volume i: Unpriviledged isa – document version 20191213*, Dec. 2019.

[41] R. Cytron and P. G. Loewner, "An automatic overlay generator," *IBM Journal of Research and Development*, vol. 30, no. 6, pp. 603–608, 1986. DOI: 10.1147/rd.306.0603.

[42] R. J. Pankhurst, "Operating systems: Program overlay techniques," *Commun. ACM*, vol. 11, no. 2, 119–125, 1968, ISSN: 0001-0782. DOI: 10.1145/362896.362923.

[43] M. A. Baker, A. Panda, N. Ghadge, A. Kadne, and K. S. Chatha, "A performance model and code overlay generator for scratchpad enhanced embedded processors," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010, pp. 287–296.

[44] C. Jang, J. Lee, B. Egger, and S. Ryu, "Automatic code overlay generation and partially redundant code fetch elimination," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 2, pp. 1–32, 2012.

[45] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, (Finland), M. Aksit and S. Matsuoka, Eds., ser. Lecture Notes in Computer Science, vol. 1241, Springer-Verlag, Jun. 1997, pp. 220–242.

[46] A. Popovici, T. Gross, and G. Alonso, "Dynamic weaving for aspect-oriented programming," in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD '02)*, (Enschede, The Netherlands), G. Kiczales, Ed., ACM Press, Apr. 2002, pp. 141–147.

[47] R. Tartler, D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk, "Dynamic aspectc++: Generic advice at any time," in *Proceedings of the 2009 Conference on New Trends in Software Methodologies, Tools and Techniques (SoMeT '09)*, (Prague, Czech Republic), H. Fujita and V. Marík, Eds., ser. Frontiers in Artificial Intelligence and Applications, Amsterdam, The Netherlands: IOS Press, 2009, pp. 165–186, ISBN: 978-1-60750-049-0. DOI: 10.3233/978-1-60750-049-0-165.

[48] V. Rodriguez, A. Duenas, and E. Stupachenko, *Function multi-versioning in gcc 6*, Jun. 2016. [Online]. Available: https://lwn.net/Articles/691932/ (visited on 01/10/2023).

[49] A. C. Inc., *Creating fat binary programs*, 1997. [Online]. Available: https://developer.apple.com/library/archive/documentation/mac/runtimehtml/RTArch-87.html (visited on 01/11/2023).

[50] A. Singh, *Mac OS X Internals: A Systems Approach: A Systems Approach*. Addison Wesley, 2016, ISBN: 0134426541.

[51] *Fatelf: Universal binaries for linux*. [Online]. Available: https://icculus.org/fatelf/ (visited on 01/11/2023).

[52] B. Wilkinson, *Something common about ms-dos and cp/m*, 1999. [Online]. Available: https://www.heco.wxwilki.com/commscpm.html (visited on 01/11/2023).

[53] A. Tevanian, M. DeMoney, K. Enderby, D. Wiebe, and G. Snyder, *Method and apparatus for architecture independent executable files*, 1995. [Online]. Available: https://patents.google.com/patent/US5432937A/en.

[54] A. Tevanian, M. DeMoney, K. Enderby, D. Wiebe, and G. Snyder, *Method and apparatus for architecture independent executable files*, 1997. [Online]. Available: https://patents.google.com/patent/US5604905/en.

[55] S. K. Cha, B. Pak, D. Brumley, and R. J. Lipton, "Platform-independent programs," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, Chicago, Illinois, USA: Association for Computing Machinery, 2010, 547–558, ISBN: 9781450302456. DOI: 10.1145/1866307.1866369.

[56] B. et al., *Pe format - win32 apps*, en-us. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 01/11/2023).

A Artifact Appendix

Abstract

This artifact includes all tools and a documentation to run the evaluation for multivariant ELFs, a binary-level, language-agnostic approach to semi-dynamic variability. It details how to run and modify four case studies using the provided artifact package, which includes benchmark scripts, a virtual disk file and the MMView kernel and its initrd. Users run the virtual machine via QEMU and execute scripts, which allow to generate and display benchmark data for each case-study. The four case studies are: (a) MariaDB ASan, (b) SQLite assertions, (c) Heterogeneous-ISA thread-pool, and (d) memcached profiling to prove the wide applicability of MELFs. Each case study includes instructions on running, exporting results, and modifying the benchmark. Next to the case-studies, users can also modify the provided MELF linker, the crucial component in creating MELFs, to examine the generation and placement of variant generation.

Scope

Users investigating the MELF benchmarks are able to verify that the only dependencies and changes to-be-made to make use of MELFs in applications are: (1) Express the existence of multiple application variants inside an application-specific linker script file (2) Extend existing application code to declare and load variants. (3) Use the llvm-based MELF linker to link application modules (object files) to the final MELF executable.

For each individual case-study, users can reconstruct the claimed benefits of MELFs described within the paper, which is mainly performance isolation and increased binary size depending on the workload. All evaluators, however, need to keep in mind that running those benchmarks in a virtualized environment will not provide the same results we were able to get for our paper. Your final result highly depends on the hardware your host machines use. But the main concept of performance isolation shall be visible in each benchmark executed.

Contents

This archive provides the user with every evaluation resource needed to run our evaluation setup. Namely, this archive consists of:

- run.sh. A script that starts a virtual machine via QEMU.
- hda.qcow2. The virtual disk file of the virtual machine which includes the whole artifact evaluation, based on debian 11.7.
- linux-mmview-vmlinuz-5.15. A Linux kernel fork of version 5.15 which includes the operating system abstractions for MMViews.
- initrd-linux-mmview-vmlinuz-5.15. The corresponding initrd of the MMView kernel fork.

- README.txt. A documentation file giving a detailed explanation of every benchmark setup and how to build, modify and draw benchmark data.

To start artifact evaluation, the user has to have QEMU installed onto their execution environment and to start run.sh. After the VM booted, the user can get access to the system by logging in either as the user "user" or as "root". The user has a Makefile inside his home directory which contains a target for each benchmark to generate the data and export that data into different formats.

Hosting

The artifact evaluation archive is hosted on the domain of our institution and can be downloaded from there: `https://sra.uni-hannover.de/Publications/2023/melf-usenix-atc23/`

Requirements

Most of our artifacts do not require specialized hardware. For the heterogeneous-ISA artifact, we execute code making use of AVX512 instructions. In order to run this artifact your host machine has to support AVX512, but most of modern hardware does that by default. Otherwise, the list of requirements is:

- Modern CPU with at least 16 cores. If you have less you have to adjust the run.sh script and the benchmarks inside the VM.
- At least 8GB RAM, the more the better.

For software requirements, you need to have installed:

- KVM module installed and loaded on your host machine.
- QEMU virtualization software stack.

Users can deviate from the given requirements, but doing so requires manual modification of run.sh and the benchmark inside the virtual machine.

APRON: Authenticated and Progressive System Image Renovation

Sangho Lee
Microsoft Research

Abstract

The integrity and availability of an operating system are important to securely use a computing device. Conventional schemes focus on how to prevent adversaries from corrupting the operating system or how to detect such corruption. However, how to recover the device from such corruption securely and efficiently is overlooked, resulting in lengthy system downtime with integrity violation and unavailability.

In this paper, we propose APRON, a novel scheme to renovate a corrupt or outdated operating system image securely and progressively. APRON concurrently and selectively repairs any invalid blocks on demand during and after the system boot, effectively minimizing the system downtime needed for a recovery. APRON verifies whether requested blocks are valid in the kernel using a signed Merkle hash tree computed over the valid, up-to-date system image. If they are invalid, it fetches corresponding blocks from a reliable source, verifies them, and replaces the requested blocks with the fetched ones. Once the system boots up, APRON runs a background thread to eventually renovate any other non-requested invalid blocks. Our evaluation shows that APRON has short downtime: it outperforms conventional recovery mechanisms by up to 28×. It runs real-world applications with an average runtime overhead of 9% during the renovation and with negligible overhead (0.01%) once the renovation is completed.

1 Introduction

Ensuring the integrity and availability of an operating system is crucial to the security of a computing device which is repeatedly threatened by adversaries. Specifically, the adversaries might compromise the operating system by exploiting unpatched vulnerabilities contained in its kernel, system applications, or shared libraries and, if exists, underlying systems software like a hypervisor. Then, they would permanently tamper with the system image (or files) stored in local storage to persist their control over the device (i.e., persistent malware [24, 29, 81]) or destroy it (i.e., destructive malware [53,77,82]). The computing device is no longer available in a valid form and demands *a recovery* as soon as the corruption is recognized [51, 98, 125].

Secure boot is a mechanism to boot a system while checking its integrity [8, 13, 52, 104, 124]. A trusted bootloader—whose authenticity is ensured by cryptography and hardware-based mechanisms [9, 102, 122]—measures (i.e., calculates a cryptographic hash over) the operating system and compares the measured value with an expected value before loading and passing control to the operating system. Any mismatch between them means that the operating system is in an invalid state. Specifically, the operating system might be (a) manipulated to embed a persistent backdoor, (b) destroyed to no longer work, or (c) downgraded to run a vulnerable old version. All these invalid states require an urgent fix.

Secure boot is suitable for securing devices with *image-based management*. They consistently deploy and update devices with *read-only* immutable system images built on the server-side and maintain their integrity on devices. However, such write protection is typically enforced within the kernel and adversaries can bypass it if they compromise the kernel [46,58,87]. Thus, operating systems require secure boot to verify the system image integrity, which is straightforward as expected measurement values are consistent or updated with coordination in image-based operating systems. Many modern operating systems for containers [36,39,45,61,86,96,105], Internet of Things (IoT) and edge devices [25, 37], mobile phones [6], mixed reality headsets [74], and personal computers [1, 10, 83, 117] adopt both.

System recovery is a logical next step when secure boot has found any corruption from a system image. Numerous security systems [3,4,17,26,51,79,98,108,125] even frequently reboot and recover (or reprovision) devices to protect them against persistent security and privacy threats and failures. However, to the best of our knowledge, all existing recovery mechanisms suffer from the following problems:

- **Downtime.** If a system recovery is necessary, a computing device enters a recovery environment [18,75,94,120]. The recovery environment does not support any other regular tasks. That is, the system is *unavailable during recovery*, which is especially unacceptable to reboot-based security

systems [3, 4, 17, 26, 51, 79, 98, 108, 125] and mission-critical tasks like edge computing [12, 84].

- **Inefficiency.** A recovery is inefficient because *it does not know which files or blocks it must fix in advance*. Existing mechanisms either (a) overwrite the entire system image to the local storage [40, 50, 51, 98, 125], which not only takes long but also is bad for storage lifetime [73, 103], or (b) verify each file or block to selectively fix corrupt one [54, 91], which is slower than the former.
- **Staleness.** A recovery typically relies on a system image backup stored in the local storage [50] which can be *outdated or corrupt*. To avoid this problem, it might fetch the latest system image from a reliable source, but this downloading prolongs the overall recovery time.

In this paper, we propose APRON, a novel approach to securely and progressively renovate an operating system image on a device. Unlike existing recovery mechanisms that fully repair a corrupt system image in a recovery environment and then boot into the recovered system, APRON securely boots into the system while repairing it, minimizing the downtime. It runs in the fresh kernel context to concurrently and selectively renovate any corrupt blocks which are requested during the system boot and after the startup of the operating system.

APRON *intervenes* between applications or kernel threads and the local storage containing a system image to verify and renovate invalid blocks on demand. To verify a requested block, APRON uses a Merkle hash tree [72] which is computed against an up-to-date system image and certified by an authorized entity (i.e., an administrator or an operating system vendor). Any hash verification error implies that a requested block is invalid (i.e., corrupt or outdated). Then, APRON renovates it by retrieving a corresponding block from a reliable source such as a remote server, verifying it, and overwriting it at the correct storage location. Once the system boots up, APRON additionally runs a background thread to renovate any other non-requested invalid blocks in the end. Also, it deduplicates any redundant network transfers.

We prototype APRON for Linux. We use device mapper [95] for intervening storage access and `dm-verity` [112] for hash tree verification. We also use Network Block Device (NBD) [22] as our remote storage protocol and its client, `nbdkit` [60], for userspace operations including HTTPS.

APRON ensures short system downtime and low runtime overhead. For a recovery with a 10 GiB system image in the servers with fast (local 1 Gbit/s) and slow (remote 100 Mbit/s) networks, APRON adds at most 5 s and 32 s to the downtime, respectively. It outperforms the full recovery by up to 28× and 12× and the delta recovery by up to 120× and 23×. APRON incurs an average runtime overhead of 9% on diverse real-world application tests from the Phoronix Test Suite [89] during renovation. Once the renovation is completed, the runtime overhead becomes negligible (0.01%).

In summary, this paper makes the following contributions:

- APRON is the first system that securely and progressively renovates a device's system image. It effectively minimizes the downtime due to recovery or update.
- APRON suggests a unified way to fix various types of invalid blocks including corrupt and outdated blocks.
- APRON shows its effectiveness (i.e., short downtime and low runtime overhead) in various configurations.

The source code of our prototype is publicly available at `https://github.com/microsoft/APRON`.

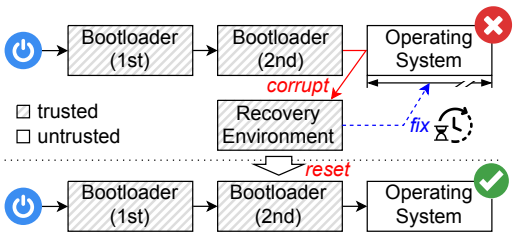


Figure 1: Secure boot with a normal recovery.

2 Background and Motivation

We will explain the background and motivation of APRON.

2.1 Secure Boot and Recovery

Secure boot ensures that a valid operating system will start to run on a device when it is powered on or reset. It has various synonyms, such as authenticated, measured, trusted, and verified boot, depending on security policies enforced or emphasized. Trusted or verified boot typically stops the boot procedure if it recognizes any verification failures, and initiates a recovery procedure. Authenticated or measured boot proceeds with the boot procedure while extending measurement values to a hardware component like Trusted Platform Module (TPM) [119] to report them to a system administrator for a later decision. In this paper, secure boot denotes trusted boot.

Figure 1 shows a procedure of secure boot with a recovery. When a device is powered on or reset, its CPU starts to execute the first-stage bootloader or boot firmware (e.g., Unified Extensible Firmware Interface (UEFI) [121], coreboot [30]) typically stored in a boot ROM. The first-stage bootloader verifies and loads the second-stage bootloader (e.g., GRUB [41], BOOTMGR [38]) stored in a specific location of local storage (e.g., EFI System Partition (ESP)). The second-stage bootloader verifies an operating system image stored in local storage. If the verification fails or a recovery has been requested, it boots into a recovery environment.

The recovery environment runs an agent program for recovery. The agent either downloads the latest system image from a known source and validates it or uses a local backup image to re-image the device [4, 18, 40, 50, 51, 98, 108, 125]. Finally, the agent reboots the device or directly loads the recovered kernel (using `kexec` in Linux [47] or Kernel Soft Reboot (KSR) in Windows [76]) and lets the kernel proceed the remaining boot procedure.

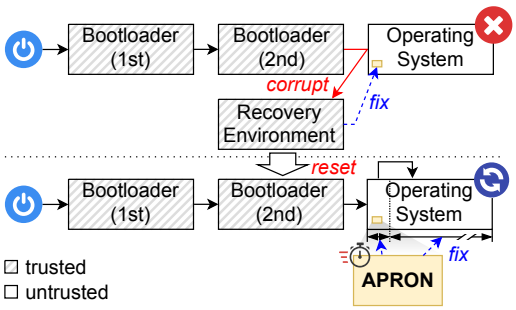


Figure 2: Secure boot with APRON.

2.2 Image-based System Management

Modern operating systems for specific use cases or casual end-users, such as container, IoT, edge, mobile phone, and personal computers [1, 6, 10, 25, 36, 37, 39, 45, 61, 74, 83, 86, 96, 105, 117] adopt image-based management to confine and simplify their deployment and update procedures. They split device storage into at least two different partitions including *read-only system partition* and *read-writable user partition*.

The system partition contains security-critical data which must not be modified, including kernel, drivers, and shared libraries. The kernel prevents any processes from modifying the partition [11, 112]. Administrators or operating system vendors generate or update a golden image for the system partition, sign it, and deploy it to devices. Instead of deploying the entire image, they can calculate and deploy the *delta* between the new and old images [34, 70, 91]. The device either fully overwrites the received image into its system partition or selectively updates it based on the delta. Later, a trusted bootloader verifies whether the system partition contains a valid, up-to-date system image before loading it.

The user partition contains casual applications and data populated by a user, which are not related to the device's critical operations. These user applications and data can be backed up by cloud storage, which is out of this paper's scope.

2.3 Motivation and Goal

Secure boot and image-based operating systems are widely deployed to real-world computing devices. However, their recovery mechanisms suffer from three important problems, motivating us to design a new approach that progressively recovers the device not only during its secure boot but also after the startup of its operating system (Figure 2).

Downtime for recovery. While the recovery environment repairs the system image, a computing device cannot conduct any regular operations that it is expected to do. That is, it cannot ensure a critical requirement, *availability*, for a long time. It is critical especially if the device leverages frequent reboots and recoveries to mitigate attacks and failures [3, 4, 17, 26, 51, 79, 98, 108, 125] or is deployed for mission-critical or time-sensitive tasks like edge computing [12, 84]. We cannot simply get rid of the recovery environment because we need a separate environment to securely trigger a recovery procedure at least. Instead, what we aim to achieve is

G1. Two-stage progressive recovery
Our approach progressively recovers the system image during and after the system boot. Its validity is ensured by a separate recovery environment.

Inefficient recovery due to unknown state. Existing recovery approaches are inefficient as they do not know which portions of the system image are corrupt in advance. Inspecting the entire image to identify invalid blocks and calculate delta takes long (§6.2). Instead, what we aim to achieve is

G2. State-aware on-demand recovery
Our approach identifies whether certain portions of the system image are corrupt and recover them on demand.

Insecure recovery due to stale image. Existing recovery approaches rely on a system image backup stored in local storage to recover the system image, which might be outdated or corrupted by adversaries. Fetching the latest image from a reliable source is a viable solution, but it is slow even if the image is compressed. Instead, what we aim to achieve is

G3. External up-to-date recovery
Our approach fetches authenticated portions of the system image on demand from a reliable external source.

3 Threat Model and Assumption

We consider how to recover a computing device from a remote adversary who can compromise the device's operating system including its kernel and system binaries as well as the underlying systems software (e.g., hypervisor and host operating system) if exists. The adversary can bypass the attack detection or prevention mechanisms for the device using unknown attack vectors, enabling initial compromise. After they compromise the system, they tamper with its storage to persist their control over it or corrupt it [24, 29, 53, 77, 81, 82].

We assume that the adversary cannot tamper with the boot firmware like other bare-metal recovery or reprovision systems [17, 51, 79, 108, 125]. Existing hardware components (e.g., boot ROM [66], TPM [119], and security coprocessors [9, 102, 122]) protect the boot firmware and its configuration data including the public key certificate of the authorized entity (i.e., administrator or operating system vendor) and a signed system version number for rollback prevention. We assume that a user or an administrator can recognize system compromise (e.g., by monitoring its external behaviors) and recover the system by forcefully rebooting it [14, 40, 118, 125] to let the boot firmware initiate a recovery procedure. In addition, the authorized entity generates and signs system images and metadata as well as operates servers for device management.

4 Design

In this section, we explain the design of APRON. It consists of (a) *storage layer* to authentically and progressively renovate

the system partition during the system boot and after the startup of the operating system, (b) *server* to maintain and deploy valid, up-to-date system images, (c) *client* to fetch specific portions of the system image and deliver them to the storage layer for renovation, and (d) *recovery environment* to initiate APRON.

Storage layer. The APRON storage layer is responsible for on-demand renovation of invalid blocks (§4.2), background prefetching (§4.3), and deduplication (§4.4). All these tasks are securely performed based on a Merkle hash tree computed over a valid, up-to-date system image.

Server and client. The APRON server manages and updates operating system images, generates metadata for them (e.g., a Merkle hash tree), signs them, and deploys them to clients. The APRON client establishes a secure session with the deployment server to fetch data blocks based on the APRON storage layer's requests and deliver them to it (§4.5).

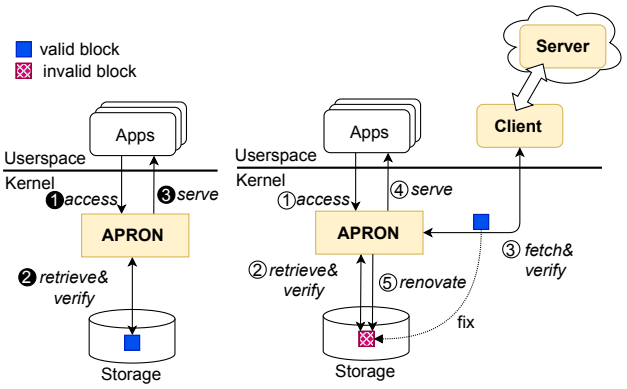
Recovery environment. The APRON-aware recovery environment prepares a minimal environment to initiate an operating system with APRON including operating system kernel and APRON metadata (§4.6).

4.1 Initialization

APRON is integrated into the operating system kernel and can be configured via boot parameters and APRON metadata. The APRON metadata includes a root hash value concatenated with a version number and Merkle hash tree calculated over the system partition. This metadata is prepared and signed by the APRON server (§4.5). APRON stores critical system files (e.g., kernel and device drivers) in the system partition and APRON metadata in a separate partition (to avoid circular dependency when calculating a hash tree). The APRON-aware recovery environment verifies this setting (§4.6).

During the system boot, APRON updates and verifies the APRON metadata. In particular, APRON attempts to download the latest APRON metadata from the APRON server. Then, it verifies the metadata's signature using the authorized entity's public key and checks whether the metadata's version number is greater than or equal to the reference version number. Both public key and reference version number are secured along with the boot firmware (§3). If the downloaded metadata is new and valid, APRON replaces the locally stored one with it and proceeds the system boot. If the version number in the new metadata is greater than the reference version number in the secure storage, APRON monotonically increases the reference version number accordingly as it means there is a legitimate update. Otherwise, APRON discards the downloaded one and proceeds the system boot with the local metadata.

When the operating system sets up a root filesystem, APRON prevents it from directly using the system partition. Instead, APRON places a *layer* (i.e., a virtual storage device or partition) over the system partition and makes the operating system use the storage layer as a read-only root filesystem. This layer allows APRON to intercept any accesses to the system partition, verify the individual accesses, and selectively fix the corresponding blocks using the Merkle hash tree.



(a) Valid access (b) Invalid access

Figure 3: On-demand renovation. Access to a valid block is verified and served using local storage (❶–❸). Access to an invalid block is served with the help of a remote server (①–⑤).

4.2 On-Demand Renovation

If any process or thread (except for APRON itself) requests a portion of the system partition, APRON transparently provides valid data either as they are or after renovating them. Figure 3 shows the overall on-demand renovation procedure. When a userspace application or kernel thread attempts to read data contained in the system partition via a filesystem or block device interface (❶), APRON retrieves a corresponding disk block expected to contain the requested data and verifies it using the Merkle hash tree (❷). If the retrieved block is valid (Figure 3a), APRON provides it to the requester (❸). To avoid repetitive storage retrieval and validation (i.e., to skip ❷), APRON maintains retrieved read-only blocks in an in-kernel cache and serves them to requesters later without re-validation until the blocks are evicted from the cache. Also, APRON identifies whether a requested block is a *zero block* (i.e., a data block filled with zeros) by checking the hash tree (i.e., a corresponding leaf node). In that case, it quickly provides a zero buffer to the requester without accessing the storage device at all. These two performance optimization methods are motivated by `dm-verity` [112].

If the retrieved block is invalid (Figure 3b), APRON fetches a corresponding block from the deployment server via the client and then verifies it (③). If the fetched block is valid, APRON provides it to the requester (④) to proceed with its execution immediately. Then, APRON overwrites the content of the fetched block into the corresponding location at the local storage device (⑤). If the fetched block is invalid due to network error, server error, or some other reason, APRON retries the block fetching up to a predefined number of times. If APRON fails to obtain the block eventually, it reboots the device to re-initiate the renovation.

In addition, APRON concurrently retrieves the same block

from the local and remote storage (i.e., conduct ② and ③ in parallel) to effectively hide network overhead. APRON activates this concurrent fetching only if it confirms that the storage device is corrupt (i.e., it finds at least one invalid block during previous storage accesses) to avoid sending meaningless network requests.

4.3 Background Prefetcher

The on-demand renovation might fail if the network connection between a device and the deployment server becomes unreliable or slow during execution. To avoid such failures, APRON has *background prefetcher* that is a kernel thread to detect and renovate invalid blocks of the system partition in advance. Background prefetcher only inspects unidentified blocks which exclude verified or renovated blocks and zero blocks according to the hash tree. To support the former, APRON maintains a verified block bitmap and updates its bits when certain blocks are verified or renovated by either the on-demand renovation or background prefetcher.

Background prefetcher inspects unidentified blocks if it does not interfere with storage accesses from other applications or kernel threads. Specifically, it wakes up if there is no in-flight access to the local storage device, checks and renovates a limited number of unidentified blocks, and sleeps.

Background prefetcher detects consecutive invalid blocks and renovates them together (i.e., a batch renovation) to improve the renovation throughput. It is different from the on-demand renovation which repairs each urgently required block with low latency. When background prefetcher wakes up, it inspects the system partition from the first unidentified block according to the verified block bitmap to detect the first invalid block. Next, it inspects the following blocks until it encounters a valid block or the number of inspected blocks exceeds a threshold, resulting in a sequence of consecutive invalid blocks. Then, it fetches corresponding blocks from the remote storage together, verifies them, overwrites them to the local storage device for batch renovation, and updates the verified block bitmap accordingly. It uses an exponential backoff algorithm to dynamically adjust the threshold.

Disconnection. If background prefetcher has fully inspected the entire system partition, APRON can be completely disconnected from the deployment server until the device gets reset or it needs to renovate or update the system image (§4.6).

4.4 Deduplication

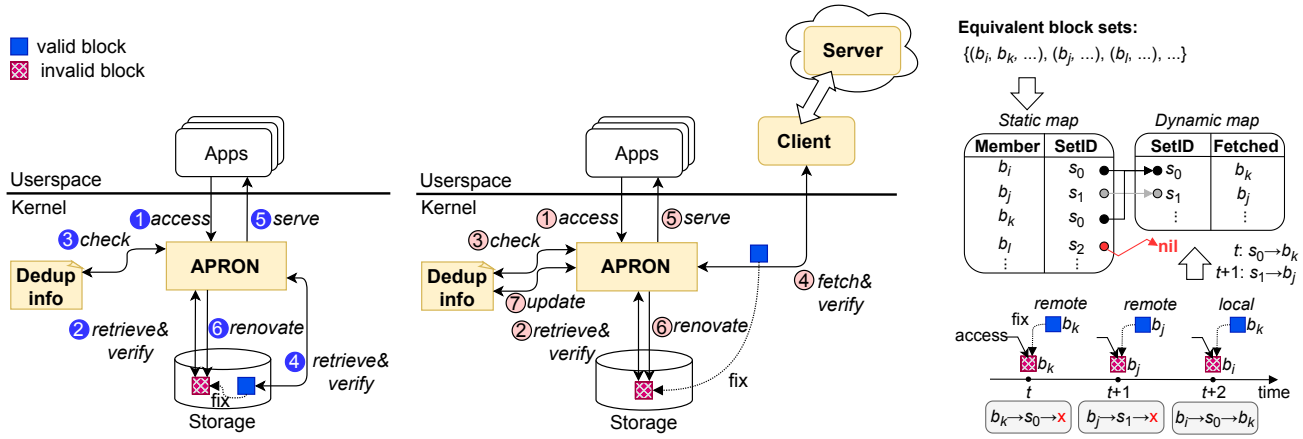
APRON might repetitively fetch equivalent blocks from the deployment server for renovation, which meaninglessly stresses both server and network. APRON avoids it using a data deduplication technique. APRON fetches a corresponding block from the server to renovate an invalid block only if it is unique or no other equivalent block of it has been fetched. If not, APRON uses the fetched equivalent block (in the local storage) for renovation. The APRON server creates deduplication metadata representing equivalent block sets in the system image. Specifically, the APRON server analyzes the system image (or the leaf nodes of its hash tree) to find equivalent blocks with the same content, forms sets by grouping equivalent blocks, assigns a unique identifier to each set, and adds this information to the deduplication metadata. The APRON server deploys and maintains the deduplication metadata together with the hash tree. To minimize the metadata size, APRON excludes unique blocks and zero blocks from it.

On the device, APRON maintains deduplication information consisting of two data structures for maintaining equivalent block sets and tracking whether and which block belonging to each set has been fetched, respectively. During initialization, APRON retrieves equivalent block set information from the deduplication metadata and constructs a *static block map* which associates each non-unique block (member) with its set identifier ($member \mapsto setID$). Later, APRON confirms whether a block is unique by looking up the static block map. If APRON fetches any non-unique block belonging to an equivalent block set for the first time (during the on-demand or background renovation), it adds this information to a *dynamic block map* to reversely associate the fetched block's set identifier with the fetched block ($setID \mapsto fetched$). This dynamic block map allows APRON to serve a duplicated block request using an equivalent block stored in the local storage.

Figure 4 shows the on-demand renovation with deduplication. The deduplication neither affects access to any valid blocks nor any invalid but unique blocks. Thus, we do not re-explain them (refer to §4.2). Instead, we focus on access to an invalid non-unique block that might have an equivalent block fetched and stored in the local storage device. If APRON confirms that a requested invalid block is not unique and its equivalent block is stored in the local storage device according to the deduplication information (i.e., the dynamic block map) (Figure 4a ❸), APRON retrieves the equivalent block from the local storage device and verifies it (❹). If the retrieved equivalent block is valid, APRON provides it to the requester (❺) and fixes the invalid local block (❻). In addition, it skips ❹ if the equivalent block is in the cache.

If the requested invalid block has no fetched equivalent block or the retrieved equivalent block is invalid (Figure 4b), APRON fetches a corresponding block from the deployment server and verifies it (④), provides it to the requester (⑤), and renovates the invalid local block (⑥). APRON updates the deduplication information (⑦) by adding the requested block as a fetched one to the dynamic block map.

Figure 4c depicts how deduplication works. In the beginning, APRON constructs a static block map using equivalent block sets and an empty dynamic block map. At time t, a thread requests an invalid and non-unique block b_k with the set ID of s_0. APRON fetches a corresponding block from the deployment server since the dynamic block map has no entry for s_0 and adds $s_0 \mapsto b_k$ to the dynamic map. At time $t+1$, a thread requests b_j with the set ID of s_1. Again, APRON fetches a corresponding block from the server and adds $s_1 \mapsto b_j$ to



(a) Invalid access with fetched equivalent block (b) Invalid access without fetched equivalent block (c) Deduplication management

Figure 4: On-demand renovation with deduplication. Access to an invalid block with a local equivalent block is served and renovated using the equivalent block (❶–❻). Access to an invalid block without a local equivalent block is served with the help of a remote server while renovating the requested block and updating the deduplication information (①–⑦).

the dynamic map. At time $t+2$, a thread requests b_i with the set ID of s_0. This time, APRON retrieves b_k from the local storage to renovate b_i because the dynamic map has $s_0 \mapsto b_k$.

Background prefetcher with deduplication. Background prefetcher prioritizes unique or never-fetched invalid blocks which must be remotely renovated over other invalid blocks which can be locally fixed. To this end, it treats invalid non-unique blocks as semi-valid if their equivalent blocks have been fetched and does not renovate them urgently. At the end of storage inspection, it spawns another kernel thread, *duplicator*, to renovate all semi-valid blocks using their local equivalent blocks. Like background prefetcher, duplicator wakes up if there is no in-flight access to the local storage device, renovates a single set of consecutive blocks, and sleeps.

4.5 Server and Client

The APRON servers are responsible for two major tasks: generating and maintaining operating system images as well as their associated metadata; and deploying them to the APRON client running in computing devices via a secure channel.

Management server. The management server is a trusted server operated by administrators or operating system vendors. It generates APRON metadata when a system image is newly generated or updated. First, it analyzes the system image to zero out unallocated data blocks—to handle sparse images—and then calculates a hash tree over the image. Next, it generates deduplication metadata (i.e., figures out equivalent block sets) using the hash tree's leaf nodes. Then, it monotonically increases the hash tree version number, and signs a concatenation of the root hash value and version number. Lastly, it stores the system image and APRON metadata in a dedicated place to be accessible to the deployment server.

Deployment server. The deployment server provides system images and APRON metadata to APRON devices. It interacts with devices via secure channels (i.e., TLS) to prevent external entities from manipulating or eavesdropping on communication. To handle data block requests, it uses a remote storage protocol (e.g., iSCSI, NBD) or a regular data transfer protocol (e.g., HTTP, FTP). The former lets it efficiently and transparently handle requests but is bad for portability. The latter introduces overhead including bloated packet headers and extra translation but ensures portability and proximity (e.g., with Content Delivery Network (CDN)). They respectively have pros and cons, and which one we are expected to choose one of them according to system configurations.

Client. The APRON client is a userspace service that interacts with the deployment server. It establishes a secure channel with the server and uses either a remote storage protocol or a regular data transfer protocol to receive data blocks.

Unlike the management server, APRON does not trust both the deployment server, which could be operated by a third party like a CDN, and the userspace client, which might be compromised by an attacker. The in-kernel APRON storage layer always verifies received blocks with the versioned APRON metadata signed by the management server. Even if the deployment server or the userspace client arbitrarily tampers with or downgrades the system image or metadata, the in-kernel APRON layer identifies such attempts based on signature and version mismatch.

4.6 First-Stage Recovery and Update

APRON can start to work only if a device has critical system files (i.e., kernel and device drivers for storage and network) and APRON metadata in valid forms. APRON relies on a first-stage recovery environment to ensure it. If any of them are invalid, the trusted bootloader enters the recovery environment to obtain their latest versions from the deployment server. A conventional APRON-unaware recovery environment would fully download both critical system files and APRON meta-

data from the deployment server via HTTPS. In contrast, an APRON-aware recovery environment only needs to download the metadata via HTTPS while selectively renovating invalid portions of the critical system files via APRON. It is possible because a recovery environment typically shares the same (or minimized) kernel with the main operating system, so we modify its kernel to incorporate APRON.

Scheduled update. APRON works as an update mechanism for non-compromised operating systems. If APRON recognizes an update during system execution, it stages updated APRON metadata on the APRON partition. Through a reset, the recovery environment replaces APRON metadata with the staged one. Finally, APRON progressively updates the system during its execution. Unlike existing update mechanisms [5,7,16], APRON does not need to reserve extra storage to temporarily store a (potentially large) update file and apply multiple update files in a proper order.

5 Implementation

In this section, we explain how we develop APRON for Linux.

Initialization. We use `initramfs` to configure and initialize the APRON storage layer as the root filesystem. Our `initramfs` checks whether the signature and version number of a given root hash value are valid using the public portion of our signing key and reference version number we provision to TPM NVRAM indexes. If they are valid, it initializes the APRON storage layer and mounts it at a specific point. Further, it creates a `tmpfs` filesystem to use it as a writable overlay for the storage layer using `overlayfs` [23] to support applications that only work with a writable root filesystem. Lastly, it sets up the overlaid storage layer as the root filesystem. We store the `initramfs` in the system partition, so it is secured by APRON as well.

Storage layer. We implement the APRON storage layer as a loadable kernel module written in approximately 1,200 lines of C code on Linux kernel version 5.11. The storage layer prototype consists of two virtual block devices representing local and remote block devices, respectively.

The local virtual block device intervenes with any access to the system partition and is exposed as a regular block device to the outside (to work as the root filesystem). It is based on the device mapper framework [95]. The storage layer verifies all accesses to the local block device using a Merkle hash tree based on `dm-verity` [112]. It also spawns background prefetcher as a kernel thread to inspect the local block device while maintaining and using the deduplication information. If the storage layer finds any corrupt blocks from the local block device, it renovates them by copying corresponding blocks from the remote virtual block device to the local virtual block device while verifying them using the hash tree. That is, it securely makes the content of the local block device equivalent to that of the remote block device. This approach also allows APRON to use a local backup device for renovation instead of a remote device if the network condition is bad or a new system image is buggy. Background prefetcher uses `kcopyd` [114] to efficiently copy a sequence of data blocks between block devices. In addition, APRON's every storage access is cached by the `dm-bufio` interface.

The remote virtual block device interacts with the deployment server via the APRON client based on NBD [22]. We use NBD because it is easy to configure (both in client and server) and has a small code base. If needed, APRON can work with other advanced remote block storage such as Ceph [123] for better efficiency, reliability, and scalability.

Server. The APRON server has a program to identify equivalent block sets written in 170 lines of Rust code, Bash scripts to automate the creation, management, and deployment of system images and APRON metadata, and other server applications. It uses `zerofree` [126] to zero out the unallocated blocks of system images, `veritysetup` [112] to calculate Merkle hash trees over them, and `openssl` [115] to sign the root hash value concatenated with a version number. Also, it uses `nbdkit` [60] and `lighttpd` [65] to operate an NBD server with TLS and an HTTPS server, respectively.

Client. We prototype the APRON client using `nbd-client` [32] and `nbdkit` [60]. It uses `nbd-client` to connect to the APRON server via NBD over TLS and configure this session as the storage layer's remote block device. Also, to interact with the APRON server via HTTPS, it uses `nbdkit` to spawn a device-local NBD server backed by the HTTPS server and lets `nbd-client` connect to this local NBD server (via a Unix domain socket). `nbdkit` relies on its `curl` plugin to fetch specific portions of system image files from the HTTPS server using HTTP `range` requests.

Recovery environment. The APRON recovery environment is based on the Linux kernel with APRON and `initramfs`. It includes `curl` [107] to download APRON metadata from the APRON server, and both `nbd-client` and `nbdkit` to renovate essential system files (i.e., kernel and `initramfs`) in the system partition on-demand. Unlike the main operating system, we decide not to spawn background prefetcher in the recovery environment because it only runs for a short amount of time. In addition, it contains `kexec` [47] to directly load the renovated operating system's kernel.

Bootloading. We decide not to manipulate the first-stage bootloader (UEFI [121]) of our computing device (e.g., replace it with coreboot [30] or replace its platform key with our own key [52]) because it is too intrusive and does not affect the core functionalities of APRON. Instead, we rely on the current UEFI-based Linux boot procedure that securely loads the second-stage bootloader, GRUB [41], signed by Linux vendors (i.e., Canonical in our case) through Shim [68] signed by Microsoft. We modify GRUB's configuration to make it load either the operating system with APRON or the APRON recovery environment.

6 Evaluation

We evaluate APRON by answering the following questions:

- **RQ1.** Does APRON ensure short system downtime when it needs to renovate the system during boot? (§6.2)
- **RQ2.** How much overhead does APRON add to other workloads during renovation? (§6.3)
- **RQ3.** What is the network usage of APRON for renovation? (§6.4)
- **RQ4**. Does APRON complete renovation within a reasonable time? (§6.5)

6.1 Setup

Device. We use a desktop computer featuring an Intel Core i5-8500 CPU (six cores) at 3 GHz, 8 GiB of RAM, and 1 TB of PCIe 3.0 NVMe SSD as an APRON device.

Server. The APRON device frequently downloads small data packets to selectively renovate the system image, subject to network performance. To evaluate it, we use two APRON servers with fast and slow network configurations. The fast-network server is a mini computer connected to the 1 GbE switch that the APRON device is also connected to. It features an Intel Pentium Silver J5005 CPU (four cores) at 1.5 GHz, 8 GiB of RAM, and 500 GB of SATA SSD. The slow-network server is a Virtual Machine (VM) in Microsoft Azure. It features two Intel vCPUs at 2.6 GHz, 8 GiB of RAM, and 30 GiB of Premium SSD. According to `netperf` [49], the median TCP latencies between the device and two servers are 0.24 ms and 5.45 ms, and the TCP throughputs between them are 934 Mbit/s and 931 Mbit/s, respectively. We additionally throttle the slow-network server's bandwidth to 100 Mbit/s to evaluate low-throughput cases.

Configuration. We install Ubuntu Server 20.04 on the device while replacing its kernel and modules with ours, computing a hash tree over the system partition, and changing its GRUB configuration. We do not use existing image-based operating systems because they are highly customized for specific platforms (e.g., VM, mobile phone). We reserve 10 GiB for a device's system partition formatted with `ext4`. Ubuntu Server 20.04 occupies 5.5 GiB of the system partition. It becomes 1.6 GiB with `gzip`. We also reserve 100 MiB to store the APRON metadata. We use 4 KiB as the data and hash block size and SHA-256 for constructing the hash tree, and RSA-4096 to sign the root hash. The sizes of the hash tree and deduplication metadata are 81 MiB and 1.6 MiB, respectively. We install Ubuntu Server 20.04 to our servers. We repeat all experiments at least 10 times and report their average values except for benchmark tools with internal repetitions (§6.3) and an experiment with deterministic results (§6.4).

6.2 System Downtime (RQ1)

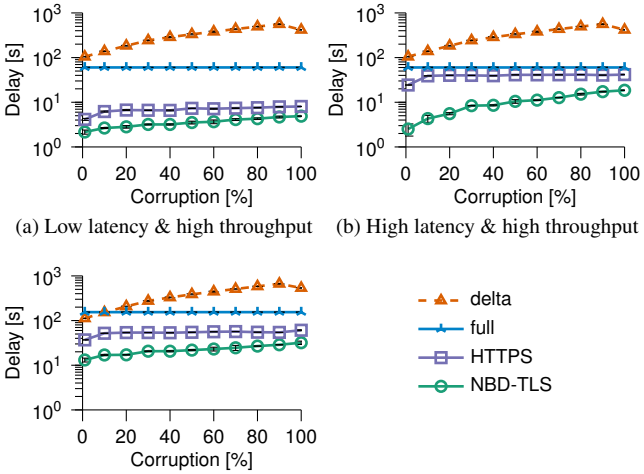
To minimize system downtime, APRON boots into a system while renovating its invalid blocks requested during the boot. We compare it against existing recovery mechanisms which repair all invalid blocks before the system boot. In general, both attacks and legitimate updates change a portion of the system image (which will be explained later), but what and how many blocks they will change are unpredictable. Thus, we randomly corrupt 1%–100% of the system partition for evaluation (i.e., zero some of or all its 4 KiB blocks). Whether we use zero or non-zero blocks does not affect APRON's performance (§6.6). We measure the delay solely introduced by APRON; that is, we exclude any other delays due to hardware initialization, bootloader loading and execution, and operating system loading, which are ∼16 s in total on our device, because they are independent of APRON. We use `systemd-analyze` [64] for this measurement. In addition, we omit the case without corruption because APRON does not delay it.

Full recovery. For comparison, we implement a full recovery mechanism like existing mechanisms [4, 40, 50, 51, 98, 108, 125]. In a recovery environment (i.e., before the system boot), an agent downloads the compressed image via HTTPS while concurrently decompressing it to the local storage. Then, it boots into the recovered system. We omit additional image validation because we trust TLS. The recovery takes ∼60 s (high throughput) and ∼154 s (low throughput), which is 4× and 10× longer than the system boot time. It is independent of the corruption ratio and marginally affected by network latency.

Delta recovery. We implement a delta recovery mechanism using `rdiff` [91] that the SWUpdate project [16] uses. `rdiff` consists of (a) signature computation, (b) delta computation, and (c) patch adoption. Delta update is efficient because it can pre-compute (a) and (b) on the server-side, but the delta recovery cannot leverage such pre-computation (details are in Appendix B.) In total, it takes 105–561 s (high throughput) and 112–665 s (low throughput), which is 7–35× and 7–42× longer than the system boot time. It depends on the corruption ratio and is marginally affected by network latency. The delta recovery is slow, but it reduces network traffic (31 MiB–1.6 GiB) and unnecessary storage writes.

APRON. Figure 5 shows APRON's boot-time delay while varying the network latency, network throughput, and corruption ratio. The delay is 2.2–4.9 s (NBD over TLS) when latency is low and throughput is high, which is 14%–31% of the boot time. It becomes 0.2–1.2× longer than the boot time when latency increases and 0.8–2.0× longer than the boot time when latency increases and throughput decreases. They are up to 27.8×, 23.9×, and 11.8× shorter than the full recovery, and up to 119.8×, 41.6×, and 23.1× shorter than the delta recovery, respectively. In addition, as expected, APRON with HTTPS is 1.6–9.7× slower than APRON with NBD over TLS due to extra translations.

Summary. APRON ensures short downtime because it apparently repairs the blocks required to boot the system first and the remaining blocks later once the system boots up. It outperforms existing mechanisms especially when (a) the network latency is low, (b) the network throughput is high, and (c) the number of invalid blocks is small. These conditions are satis-



(a) Low latency & high throughput (b) High latency & high throughput

(c) High latency & low throughput

Figure 5: Boot-time delay comparison between APRON and other recovery mechanisms.

fied in practice. The network performance is improving. Also, the number of invalid blocks is generally small in both attack and update cases. For example, a persistent backdoor usually consists of a few small executables [29, 81]. We further analyze how Flatcar Container Linux [39] (an image-based operating system) and Fedora Cloud [93] (a pre-installed cloud image) change over their releases. Across 36 Flatcar releases for two years (version 2512.2.0 to 3139.1.2 for QEMU x64), 9.7%–12.2% of their 8.5 GiB images change. Also, across 11 Fedora Cloud releases for five years (version 26 to 36 for QEMU x64), 12.5%–21.7% of their 4.5 GiB images change. Thus, APRON is effective in decreasing the system downtime.

6.3 Runtime Overhead (RQ2)

APRON verifies and renovates the (remaining) system partition during system execution. We evaluate how its **verification** and **renovation** processes affect the runtime performance of other workloads using benchmark programs. We ensure the benchmark programs terminate before renovation is completed. Otherwise, they run on the recovered system which hides renovation overhead. To this end, we prolong renovation using a fully corrupt system image and the slow-network server and select benchmark programs which take shorter than the renovation (§6.5). We install them in another system image copy and execute them via APRON. We compare APRON to a **pristine** environment with the same hardware and operating system except that it runs an unmodified kernel.

Microbenchmark. Figure 6 shows LMbench [71] system call execution times normalized to those from the pristine environment. Both in a verification condition and during the renovation, the overheads of the APRON device over the pristine device are 0%–40%. As expected, APRON affects system calls related to filesystem and network (e.g., `stat`, `fstat`, `open`/`close`, UNIX socket). On a geometric mean, the overheads of the APRON device in a verification condition and during the renovation are 7% and 8%, respectively.

The APRON device might download a lot of data from the deployment server to renovate the system partition, so it might affect the network performance of other applications. We use LMbench's network throughput evaluation results to identify whether and how APRON affects LMbench's network throughputs (Figure 7). In a verification condition, the network throughput of LMbench on the APRON device is 3.6% (geometric mean) lower than that on the pristine device. During the renovation, the network throughput of LMbench on the APRON device is 11.8% (geometric mean) lower than that on the pristine device. Consequently, APRON's renovation noticeably affects the network throughput of other applications only during renovation. In addition, if we turn off APRON's verification once the renovation is completed, the overhead becomes almost zero.

Macrobenchmark. We evaluate APRON with 11 real-world application tests from Phoronix Test Suite [89] (Figure 8). The overheads of APRON during renovation over the pristine environment are 1.9%–21% (geometric mean: 9%). As expected, renovation affects I/O-intensive workloads (e.g., 7-Zip, Apache, and Memcached) whereas less affects compute-intensive workloads (e.g., Crafty, FLAC, and PyBench). Without renovation (i.e., the verification condition), APRON's overhead over the pristine environment is negligible (0.01%).

6.4 Network Usage (RQ3)

During renovation, APRON fetches blocks required for recovery from the deployment server. We evaluate this network usage. We use the randomly corrupt system images again (§6.2). We count the number of bytes the server sends to the APRON device at the server (using `lighttpd` logs) while enabling APRON's optimization (i.e., zero block ignorance and deduplication). This network usage is independent of network latency and throughput.

Figure 9 shows how many bytes APRON downloads from the server while varying the corruption ratio. It downloads 44.6%–94.0% of the corrupt blocks, which are only 1.1–2.9× larger than corresponding `rdiff` deltas. APRON's optimization is effective especially when the number of invalid blocks is large. This is because it increases the probability that multiple invalid blocks are equivalent such that no repetitive downloads are needed due to deduplication (§4.4).

6.5 Complete Renovation Time (RQ4)

Once the system boots up, APRON performs both on-demand and background renovation. We measure how long it takes to complete the renovation when the system is idle or busy. Hastening the complete renovation is not our goal and that is why we assign a low priority to background prefetcher to minimally affect other workloads (§6.3). We compare it against the full and delta recovery to check whether it is reasonable.

Idle system. Figure 10 shows APRON's complete renovation time in an idle system while varying the network latency,

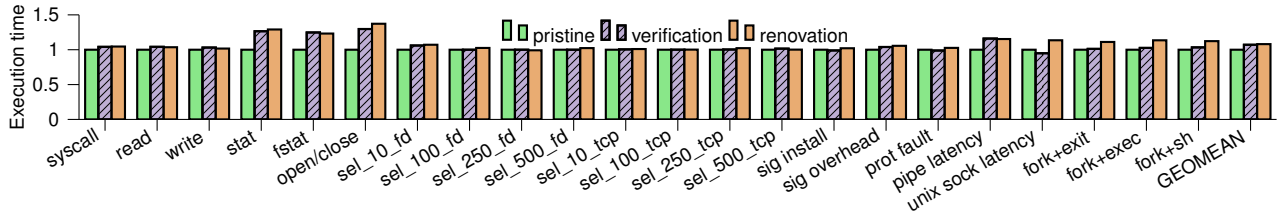


Figure 6: LMbench system call execution time normalized to the pristine cases (**pristine**: Ubuntu Server 20.04 without APRON; **verification**: APRON without a renovation process but with hash-tree verification; **renovation**: APRON with a renovation process).

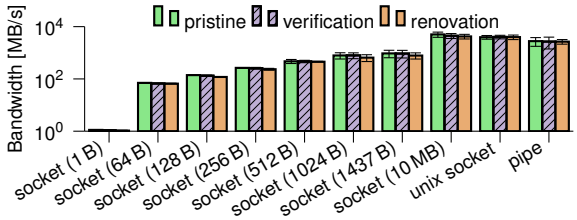


Figure 7: LMbench network throughput.

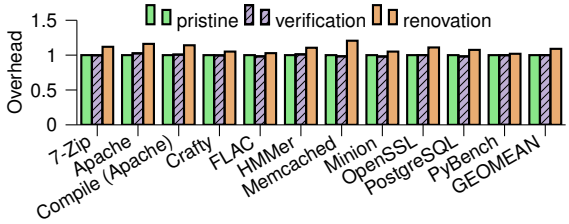


Figure 8: Normalized Phoronix Test Suite overhead.

network throughput, and corruption ratio. The complete renovation (NBD over TLS) demands 47.4–104.5 s when latency is low and throughput is high, 50.2–190.4 s when latency increases, and 84.0–504.3 s when latency increases and throughput decreases. They take 0.8–1.7×, 0.8–3.2×, and 0.5–3.3× longer than the full recovery, and 2.2–5.5×, 1.9–3.1×, and 1.1–1.4× shorter than the delta recovery, respectively. In addition, APRON with HTTPS takes 1.1–2.1× longer than APRON with NBD over TLS. APRON's complete renovation is subject to network latency and throughput because it does not benefit from bulk network transfer and compression.

Busy system. We make the system busy by running the Memcached test from Phoronix [89], which heavily contends with APRON (Figure 8), once the system boots up. The complete renovation is delayed by at most 7.0% (low latency) and 2.5% (high latency and low throughput). Overall, the renovation is moderately affected by the system's busyness. The renovation with the slow-network server suffers less from the busyness than that with the fast-network server since the slow network performance dominates the renovation overhead.

Summary. Although APRON renovates the system during its execution in the background, it finishes the renovation within a reasonable time—i.e., it is at most 3× slower than the full recovery. Further, it is comparable to or even faster than the full recovery if the network is fast and the number of invalid blocks is small. Both are satisfied in practice (§6.2).

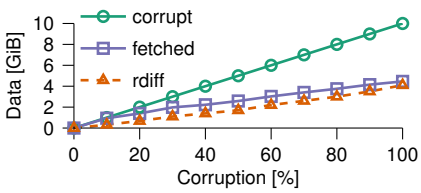
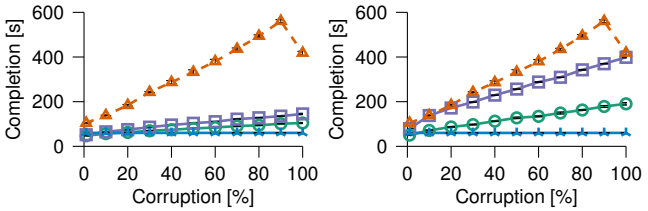
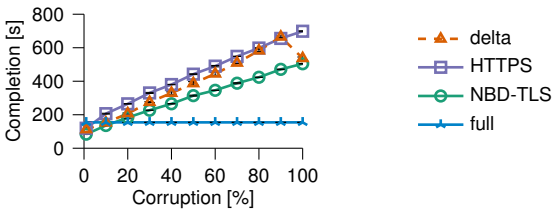


Figure 9: Network usage of APRON.



(a) Low latency & high throughput (b) High latency & high throughput



(c) High latency & low throughput

Figure 10: Complete recovery time comparison between APRON and other recovery mechanisms.

6.6 Miscellaneous

Non-zero corruption. We confirm whether we use zero or non-zero blocks to corrupt the system image does not affect APRON's performance. If a block is expected to be a zero block, APRON does not touch it regardless of corruption. If not, APRON renovates it regardless of whether it is overwritten by a zero or non-zero erroneous block. Non-zero corruption slows down the delta recovery as it complicates `rdiff` signatures, but showing its worst case is not our interest.

Deduplication. The deduplication decreases not only network usage but also renovation time to some extent. Without it, the renovation is delayed by up to 2.2% (low latency) and 9.0% (high latency and low throughput). The renovation with the fast-network server marginally benefits from the deduplication because it aims for reducing the network overhead (§6.4). In contrast, the system downtime does not benefit from the deduplication because only a few blocks are requested during the system boot.

Memory usage. APRON uses at most ∼120 MiB of additional memory when it renovates a fully corrupt system image to cache fetched blocks and maintain its data structures. This memory is reclaimed once the renovation is completed.

7 Security Analysis

In this section, we analyze the security of APRON. We focus on how APRON mitigates persistent kernel attacks. Also, we explain how APRON detects or prevents less severe but frequent userspace and network attacks.

Kernel attack. APRON efficiently recovers a computing device from advanced adversaries who have compromised the device with critical kernel vulnerabilities. APRON runs in the kernel and it does not assume any other non-standard kernel integrity protection or monitoring technologies [15, 44, 69, 78, 88, 128] to protect itself. To this end, if adversaries compromise the kernel, they can *temporarily* deactivate APRON and corrupt the system image. However, APRON eventually defeats them based on a system administrator's input: the administrator can detect a misbehaving device using device or network monitoring tools and forcefully reset it using existing, standardized techniques [40, 118, 125]. This reset eliminates in-memory exploits and activates APRON via secure boot again. If the administrator prepares an updated image fixing the exploited vulnerability, APRON prevents the adversaries from reusing the vulnerability by rapidly recovering the system with the updated image. Even if no patch is prepared, APRON causes hardship to the adversaries since they must repeatedly compromise the operating system across forceful device resets to persist their control or system destruction. Such repetitive attack attempts are highly visible and thus can be detected and mitigated by network-level techniques [3].

Userspace attack. APRON prevents or detects userspace attacks against it through all four attack surfaces userspace attackers can access (§5): (a) filesystem containing operating system files, (b) APRON storage layer, (c) storage device storing the system image, and (d) APRON client. First, APRON prevents or detects the modification of its filesystem. APRON mounts the filesystem for the operating system as read only, preventing non-privileged attacks. Adversaries with a root privilege can remount the filesystem and modify it. However, such modifications only remain in memory and are not reflected in the underlying write-protected APRON storage layer. Second, APRON prevents the modification of its storage layer. The APRON storage layer ignores any block write requests via block IO interfaces. Thus, both non-privileged and privileged adversaries cannot modify it. Third, APRON prevents or detects the modification of its storage device. Non-privileged adversaries cannot access the storage device. In contrast, privileged adversaries can tamper with the storage device using block IO interfaces. However, the APRON storage layer identifies and reverts such manipulation based on the signed hash tree. Fourth, APRON detects a misbehaving APRON client. Privileged adversaries can compromise the userspace APRON client to deliver manipulated data to the APRON storage layer. However, APRON identifies and ignores such manipulated data based on the signed hash tree.

Network attack. APRON detects network attacks including traffic manipulation. Adversaries might tamper with the network traffic between the APRON client and deployment server to deliver a manipulated system image. However, since APRON traffic is secured with TLS, the adversaries cannot arbitrarily manipulate it unless they break TLS or compromise the deployment server. Even if they succeed, APRON drops such manipulation because it verifies fetched data using the signed hash tree.

8 Discussion

In this section, we discuss some possible alternatives to APRON's design.

Advanced deduplication and compression. APRON currently uses a simple block-based deduplication (§4.4) without network traffic compression, resulting in relatively high network usage (§6.4). APRON can reduce it using advanced deduplication techniques like content-defined chunking [31, 67, 80] and seekable compression [116], but there are two tradeoffs. First, their computational overhead is higher than block-based deduplication, increasing the overall recovery time and runtime overhead. Second, they are incompatible with the efficient, block-level hash tree [112] APRON leverages. To maintain the compatibility, APRON requires a fine-grained hash tree with complicated data structure and computational complexity. We leave balancing these tradeoffs to future work.

File-based recovery. APRON verifies and renovates the entire storage or partition as it assumes the image-based system management (§2.2). Instead, it can focus on a set of critical files if it separately maintains their root hashes using techniques like Integrity Measurement Architecture (IMA) [62] and `fs-verity` [113]. This file-based recovery potentially reduces the overhead of APRON, but it must overcome two problems. First, it requires a separate technique to recover the kernel and filesystem itself. Otherwise, it even cannot identify whether certain files exist in the device. Second, it must maintain and update a set of root hash values in a scalable and consistent manner. This is because there are many critical files depending on each other (e.g., system binaries and shared libraries). APRON is free from such problems because it is independent from the filesystem and it only maintains a single root hash value for the entire image.

Mutable data. APRON leverages and ensures the read-only property of the image-based system management, but it does not prevent users from storing any data in the device. Like other image-based operating systems, APRON can maintain a separate read-writable user partition (§2.2). Then, APRON can let users use it as a writable overlay for the storage layer (as explained in §5) or mounting it at specific writable directories (e.g., `/etc`, `/home`) [33].

Single point of failure. APRON can suffer from a single-point-of-failure problem because its recovery task relies on a server which is remote in most cases. To overcome it, APRON needs other techniques like a load balancer [21] to mitigate this problem. Especially, the HTTPS version of APRON seamlessly benefits from such load balancing (§4.5).

9 Related Work

In this section, we explain existing studies related to APRON.

Network boot. Datacenter administrators frequently provision operating systems on new or failed server machines. They use the Preboot eXecution Environment (PXE) boot [55] to make each server boot into a small operating system stored in a storage server within the same local network. This small operating system downloads a full operating system to the local storage and, finally, boots into it. However, the PXE boot neither efficiently downloads system images nor ensures any network-level security because it relies on TFTP [106]. Thus, it can only be used within a well-managed local network. To overcome these problems, iPXE and UEFI support HTTP(S) boot [56, 59]. Still, they must download an entire operating system image to local storage to boot into it unlike APRON.

Diskless boot. Administrators can configure an operating system to use network storage as its root filesystem via remote block storage protocols (e.g., iSCSI and NBD) or network file systems (e.g., NFS and Samba). It is known as diskless boot [35,48,79,97]. It makes much more sense in a data center where servers are connected through the same high-bandwidth and low-latency local network [20]. However, since this approach fully relies on network storage, it cannot avoid repetitive fetching of the same blocks from storage servers if the blocks are evicted from the cache due to memory pressure. Further, a lack of required blocks due to potential network errors can result in significant system malfunctioning. Data block caching [26, 100, 101, 109, 110] might mitigate these problems, but cached blocks can be evicted according to the cache replacement policy unlike APRON. Also, all cached blocks should be accessed via a translation layer and discarded when any recovery or update is needed. Advanced distributed file systems [2, 101, 123] can avoid some of the problems, but they have large code bases and require complicated server- and client-side configuration. Unlike them, APRON only requires maintaining a simple file or web server.

Operating system streaming deployment. An operating system streaming deployment [28,42,43,85,111] uses both local and network storage. While serving block requests from kernel threads and other applications using network storage, the operating system streaming deployment stores downloaded blocks at the corresponding locations of local storage. These stored blocks will be used to resolve further requests to avoid repetitive downloading of the same blocks. The operating system streaming deployment also copies not-yet-downloaded blocks from network storage to local storage in the background to eventually mirror the network storage to the local storage. However, unlike APRON, existing operating system streaming deployment mechanisms neither consider secure operating system deployment nor support selective renovations of invalid blocks. Thus, it must deploy the entire operating system image from scratch if it recognizes any corruption or the operating system image has been updated.

Efficient update. Updating an operating system or its kernel with minimal downtime is heavily studied [5, 7, 16, 19, 27, 63, 90, 99, 127, 130]. A/B update [5, 7, 16] has a separate partition to download an updated system image during execution and reboot into it. Live kernel patching [19, 27, 90, 130] hot fixes the kernel without rebooting it. Since live kernel patching cannot handle complicated changes (e.g., data layout), other schemes [63, 99, 127] leverage memory snapshot and soft reboot. However, all these mechanisms work only if an operating system or underlying systems software (i.e., hypervisor [99], System Management Mode (SMM) [130]) is not compromised or corrupt. For example, a privileged attacker can tamper with both A/B partitions, hinder hot-patching, or corrupt memory or storage snapshots. Thus, they should rely on recovery mechanisms APRON to fix corrupt systems.

Multi-node progressive recovery. Multi-node progressive recovery [3, 57, 92, 129] is another way to recover or update a system with minimal or zero downtime. To maximize the availability of a critical service, these schemes operate redundant copies of the same service in multiple physical or virtual computing nodes. If the service needs to be recovered or updated, they first deal with a part of the nodes while running the other part of nodes to keep alive the service. They handle the latter part of nodes after they have recovered or updated the former part of the nodes. These schemes can achieve zero downtime if at least one node is always running. However, their resource costs are high because they require multiple nodes. In addition, we note that they can benefit from APRON because, in the end, they recover or update each node—reducing node recovery or update time is important to maintain their overall fault tolerance.

10 Conclusion

APRON is a novel approach to authentically and progressively renovate an operating system image during the system boot and after the startup of the operating system, minimizing the system downtime needed for a recovery. It is especially effective for the reboot-based security systems that frequently reset and repair devices to deal with attacks and failures, and mission-critical systems which are sensitive to the downtime. APRON renovates the entire operating system image with negligible runtime overhead and small network usage.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Eric Eide, for their helpful feedback.

References

[1] A Fedora initiative. Fedora Silverblue. https://silverblue.fedoraproject.org.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[3] Hussain M. J. Almohri, Layne T. Watson, and David Evans. Misery digraphs: Delaying intrusion attacks in obscure clouds. *IEEE Transactions on Information Forensics and Security*, 13(6):1361–1375, 2017.

[4] Eduardo Alvarenga, Jan R. Brands, Peter Doliwa, Jerry den Hartog, Erik Kraft, Marcel Medwed, Ventzislav Nikov, Joost Renes, Martin Rosso, Tobias Schneider, and Nikita Veshchikov. Cyber resilience for the Internet of Things: Implementations with resilience engines and attack classifications. *IEEE Transactions on Emerging Topics in Computing*, 2022.

[5] Android Open Source Project. A/B (seamless) system updates. https://source.android.com/devices/tech/ota/ab.

[6] Android Open Source Project. Implementing dm-verity. https://source.android.com/security/verifiedboot/dm-verity.

[7] Android Open Source Project. Overview of Virtual A/B. https://source.android.com/devices/tech/ota/virtual_ab.

[8] Android Open Source Project. Verified Boot. https://source.android.com/security/verifiedboot.

[9] Apple. Apple T2 security chip: Security overview, 2018. https://www.apple.com/mideast/mac/docs/Apple_T2_Security_Chip_Overview.pdf.

[10] Apple Support. About the read-only system volume in macOS Catalina. https://support.apple.com/en-us/HT210650.

[11] Apple Support. System integrity protection. https://support.apple.com/guide/security/system-integrity-protection-secb7ea06b49/1/web/1.

[12] Atakan Aral and Ivona Brandic. Dependency mining for service resilience at the edge. In *Proceedings of the 3rd IEEE/ACM Symposium on Edge Computing (SEC)*, pages 228–242, 2018.

[13] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 18th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 1997. IEEE.

[14] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[15] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the Arm TrustZone secure world. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.

[16] Stefano Babic. Software management on embedded systems. https://sbabic.github.io/swupdate/overview.html.

[17] Yechan Bae, Sarbartha Banerjee, Sangho Lee, and Marcus Peinado. Spacelord: Private and secure smart space sharing. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2022.

[18] Adrian Baldwin, Tristan Caulfield, Marius-Constantin Ilau, and David Pym. Modelling organizational recovery. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUtools)*, 2021.

[19] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC), General Track*, pages 279–291, 2005.

[20] Nick Black. Dynamic iSCSI at scale: Remote paging at Google, 2015. Linux Plumbers Conference.

[21] Tony Bourke. *Server Load Balancing*. " O'Reilly Media, Inc.", 2001.

[22] P. T. Breuer. The network block device, 2000. https://www.linuxjournal.com/article/3778.

[23] Neil Brown. Overlay filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[24] Kevin R. B. Butler, Stephen McLaughlin, and Patrick D. McDaniel. Rootkit-resistant disks. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October 2008.

[25] Canonical Ltd. Ubuntu Core. https://ubuntu.com/core.

[26] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[27] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive Android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[28] David Clerc, Luis Garcés-Erice, and Sean Rooney. OS streaming deployment. In *Proceedings of the International Performance Computing and Communications Conference (IPCCC)*, 2010.

[29] Eric Cole. *Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization*. Newnes, 2012.

[30] coreboot. coreboot. https://coreboot.org.

[31] Landon P Cox, Christopher D Murray, and Brian D Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[32] Debian Manpages. nbd-client. https://manpages.debian.org/testing/nbd-client/nbd-client.8.en.html.

[33] Debian Wiki. ReadonlyRoot. https://wiki.debian.org/ReadonlyRoot.

[34] Matthew Endsley. bsdiff/bspatch. https://github.com/mendsley/bsdiff.

[35] Christian Engelmann, Hong Ong, and Stephen L. Scott. Evaluating the shared root file system approach for diskless high-performance computing systems. In *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing (HPCC)*, 2009.

[36] Fedora Docs. Fedora CoreOS. https://docs.fedoraproject.org/en-US/fedora-coreos/.

[37] Fedora Docs. Fedora Internet of Things. https://docs.fedoraproject.org/en-US/iot/.

[38] Tim Fisher. What is the Windows Boot Manager (BOOTMGR)?, 2020. https://www.lifewire.com/windows-boot-manager-bootmgr-2625813.

[39] Flatcar Project Contributors. Flatcar Container Linux. https://www.flatcar.org.

[40] Jessie Frazelle. Opening up the baseboard management controller. *Communications of the ACM*, 63(2):38–40, 2020.

[41] Free Software Foundation (FSF). GNU GRUB - GNU project. https://www.gnu.org/software/grub/.

[42] Luis Garces-Erice and Sean Rooney. Scaling OS streaming through minimizing cache redundancy. In *Proceedings of the 31st International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2011.

[43] Luis Garcés-Erice and Sean Rooney. Secure lazy provisioning of virtual desktops to a portable storage device. In *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing Date (VTDC)*, 2012.

[44] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2003.

[45] Google Cloud. Container-Optimized OS from Google documentation. https://cloud.google.com/container-optimized-os/docs.

[46] Google Cloud. Security overview Container-Optimized OS. https://cloud.google.com/container-optimized-os/docs/concepts/security.

[47] Vivek Goyal. kexec: A new system call to allow in kernel loading, 2014. https://lwn.net/Articles/582711/.

[48] Riccardo Gusella. The analysis of diskless workstation traffic on an Ethernet. Technical report, CALIFORNIA UNIV BERKELEY COMPUTER SYSTEMS RESEARCH GROUP, 1987.

[49] HP. The netperf homepage. https://hewlettpackard.github.io/netperf/.

[50] HP. HP Sure Recover whitepaper, 2021. https://www8.hp.com/h20195/v2/GetPDF.aspx/4AA7-4556ENW.pdf.

[51] Manuel Huber, Stefan Hristozov, Simon Ott, Vasil Sarafov, and Marcus Peinado. The lazarus effect: Healing compromised devices in the internet of small things. In *Proceedings of the 15th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Taipei, Taiwan, October 2020.

[52] Trammell Hudson. safeboot. https://safeboot.dev.

[53] IBM. What is destructive malware?, 2019. https://www.ibm.com/downloads/cas/XZGZLRVD.

[54] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. HEALED: Healing & attestation for low-end embedded devices. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2019.

[55] Intel Corporation. Preboot Execution Environment (PXE) Specification Version 2.1, 1999.

[56] iPXE. iPXE - open source boot firmware. http://ipxe.org.

[57] Genya Ishigaki, Siddartha Devic, Riti Gour, and Jason P Jue. DeepPR: Progressive recovery for interdependent VNFs with deep reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 38(10):2386–2399, 2020.

[58] Jeremy Cowan, Sai Charan Teja Gopaluni, and Vijay K Sikha. Security features of Bottlerocket, an open source Linux-based operating system, 2021. https://aws.amazon.com/blogs/opensource/security-features-of-bottlerocket-an-open-source-linux-based-operating-system/.

[59] Wu Jiaxin, Fu Siyuan, and Brian Richardson. Getting started with UEFI HTTPS boot on EDK II. https://laurie0131.gitbooks.io/getting-started-with-uefi-https-boot-on-edk-ii/content/.

[60] Richard W. M. Jones. nbdkit. https://gitlab.com/nbdkit/nbdkit.

[61] Samuel Karp. Bottlerocket: A special-purpose container operating system, 2020. https://aws.amazon.com/blogs/containers/bottlerocket-a-special-purpose-container-operating-system/.

[62] Dmitry Kasatkin, David Safford, and Mimi Zohar. Integrity measurement architecture (IMA) wiki. https://sourceforge.net/p/linux-ima/wiki/Home/.

[63] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant OS updates via userspace checkpoint-and-restart. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[64] Aaron Kili. systemd-analyze – find system boot-up performance statistics in Linux, 2018. https://www.tecmint.com/systemd-analyze-monitor-linux-bootup-performance/.

[65] Jan Kneschke. lighttpd. https://www.lighttpd.net.

[66] Xeno Kovah and Corely Kallenberg. Advanced x86: BIOS and system management mode internals SPI flash protection mechanisms, 2014. http://opensecuritytraining.info/IntroBIOS.html.

[67] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2010.

[68] Eva-Katharina Kunst and Jürgen Quade. Linux control over secure boot, 2018. https://www.linux-magazine.com/Issues/2018/206/Linux-Secure-Boot-with-Shim.

[69] Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. Ki-Mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[70] Joshua MacDonald. Xdelta. https://github.com/jmacd/xdelta.

[71] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, January 1996.

[72] Ralph C Merkle. Protocols for public key cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, April 1980.

[73] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2015.

[74] Microsoft Docs. State separation and isolation. https://docs.microsoft.com/en-us/hololens/security-state-separation-isolation.

[75] Microsoft Docs. Windows recovery environment (Windows RE), 2017. https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-recovery-environment--windows-re--technical-reference.

[76] Microsoft Docs. Kernel soft reboot in Azure Stack HCI, 2021. https://docs.microsoft.com/en-us/azure-stack/hci/manage/kernel-soft-reboot.

[77] Microsoft Threat Intelligence Center. Destructive malware targeting Ukrainian organizations, 2022. https://www.microsoft.com/security/blog/2022/01/15/destructive-malware-targeting-ukrainian-organizations/.

[78] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[79] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Charles Munson, Trammell Hudson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting security sensitive tenants in a bare-metal cloud. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[80] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, October 2001.

[81] Ramin Nafisi. FoggyWeb: Targeted NOBELIUM malware leads to persistent backdoor, 2021. https://www.microsoft.com/security/blog/2021/09/27/foggyweb-targeted-nobelium-malware-leads-to-persistent-backdoor/.

[82] National Cybersecurity and Communications Integration Center. Destructive malware, 2017. https://www.cisa.gov/uscert/sites/default/files/documents/Destructive_Malware_White_Paper_S508C.pdf.

[83] NixOS contributors. Nix: Reproducible builds and deployments. https://nixos.org.

[84] Shadi A Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Computing and Communications*, 23(4):11–20, 2020.

[85] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. Improving agility and elasticity in bare-metal clouds. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[86] openSUSE contributors. openSUSE MicroOS. https://microos.opensuse.org.

[87] Perception Point. Technical analysis of CVE-2022-22583: Bypassing macOS system integrity protection (SIP), 2022. https://perception-point.io/technical-analysis-of-cve-2022-22583-bypassing-macos-system-integrity-protection/.

[88] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13rd USENIX Security Symposium (Security)*, San Diego, CA, August 2004.

[89] Phoronix Media. Phoronix Test Suite - Linux testing & benchmarking platform, automated testing, open-source benchmarking. https://www.phoronix-test-suite.com.

[90] Josh Poimboeuf. Introducing kpatch: Dynamic kernel patching, 2014. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching.

[91] Martin Pool. rdiff(1) - Linux man page. https://linux.die.net/man/1/rdiff.

[92] Mahsa Pourvali, Kaile Liang, Feng Gu, Hao Bai, Khaled Shaban, Samee Khan, and Nasir Ghani. Progressive recovery for network virtualization after large-scale disasters. In *Proceedings of the 2016 International Conference on Computing, Networking and Communications (ICNC)*, 2016.

[93] Red Hat. Fedora Cloud. https://alt.fedoraproject.org/cloud/.

[94] Red Hat Customer Portal. 32.2. Anaconda rescue mode Red Hat Enterprise Linux 7. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/installation_guide/sect-rescue-mode.

[95] Red Hat Customer Portal. Appendix A. The Device Mapper Red Hat Enterprise Linux 7. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/logical_volume_manager_administration/device_mapper.

[96] Red Hat Software. Red Hat Enterprise Linux Atomic Host: A platform optimized for Linux containers.

[97] Paul Riddle. Automated upgrades in a lab environment. In *Proceedings of the 8th USENIX Symposium on Large Installation System Administration Conference (LISA)*, 1994.

[98] Jonas Röckl, Mykolai Protsenko, Monika Huber, Tilo Müller, and Felix C Freiling. Advanced system resiliency based on virtualization techniques for IoT devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2021.

[99] Mark Russinovich, Naga Govindaraju, Melur Raghuraman, David Hepkin, Jamie Schwartz, and Arun Kishan. Virtual machine preserving host updates for zero day patching in public cloud. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, Virtual, April 2021.

[100] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[101] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[102] Uday Savagaonkar, Nelly Porter, Nadim Taha, Benjamin Serebrin, and Neal Mueller. Titan in depth: Security in plaintext, 2017. https://cloud.google.com/blog/products/gcp/titan-in-depth-security-in-plaintext.

[103] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2016.

[104] Quentin Schulz and Mylène Josserand. Secure boot from A to Z, 2018. Embedded Linux Conference.

[105] Sidero Labs, Inc. Talos Linux. https://www.talos.dev.

[106] K Sollins. The TFTP protocol (revision 2). Technical report, STD 33, RFC 1350, MIT, 1992.

[107] Daniel Stenberg. curl. https://curl.se.

[108] Kuniyasu Suzaki, Akira Tsukamoto, Andy Green, and Mohammad Mannan. Reboot-oriented IoT: Life cycle management in trusted execution environment for disposable IoT devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

[109] The kernel development community. A block layer cache (bcache). https://www.kernel.org/doc/html/latest/admin-guide/bcache.html.

[110] The kernel development community. dm-cache. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/cache.html.

[111] The kernel development community. dm-clone. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-clone.html.

[112] The kernel development community. dm-verity. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/verity.html.

[113] The kernel development community. fs-verity: read-only file-based authenticity protection. https://www.kernel.org/doc/html/latest/filesystems/fsverity.html.

[114] The kernel development community. kcopyd. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/kcopyd.html.

[115] The OpenSSL Project Authors. OpenSSL – cryptography and SSL/TLS toolkit. https://openssl.org.

[116] The Tukaani Project. XZ Utils. https://tukaani.org/xz/.

[117] Josh Triplett. Chrome OS internals, 2014. LinuxCon Europe.

[118] Trusted Computing Group. TPM 2.0 Authenticated Countdown Timer (ACT) Command, 2019.

[119] Trusted Computing Group. Trusted Platform Module Library - Part 1: Architecture, 2019.

[120] Ubuntu documentation. LiveCdRecovery. https://help.ubuntu.com/community/LiveCdRecovery.

[121] UEFI Forum. Unified Extensible Firmware Interface (UEFI) specification, version 2.9, 2021.

[122] Kushagra Vaid. Microsoft creates industry standards for datacenter hardware storage and security, 2018. https://azure.microsoft.com/en-us/blog/microsoft-creates-industry-standards-for-datacenter-hardware-storage-and-security/.

[123] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[124] Richard Wilkins and Brian Richardson. UEFI secure boot in modern computer security solutions, 2013.

[125] Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Matoon, Rob Spiger, and Stefan Thom. Dominance as a new trusted computing primitive for the Internet of Things. In *Proceedings of the 40th IEEE*

Symposium on Security and Privacy (Oakland), San Francisco, CA, May 2019.

[126] Ron Yorston. Keeping filesystem images sparse. https://frippery.org/uml/.

[127] Anthony Yznaga. PKRAM: Preserved-over-Kexec RAM. https://lwn.net/Articles/851192/.

[128] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. SPECTRE: A dependable introspection framework via System Management Mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[129] Yangming Zhao, Mohammed Pithapur, and Chunming Qiao. On progressive recovery in interdependent cyber physical systems. In *Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM)*, 2016.

[130] Lei Zhou, Fengwei Zhang, Jinghui Liao, Zhengyu Ning, Jidong Xiao, Kevin Leach, Westley Weimer, and Guojun Wang. KShot: Live kernel patching with SMM and SGX. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

A Merkle Hash Tree

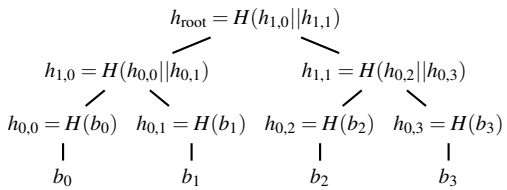


Figure 11: Merkle hash tree

A Merkle hash tree [72] is a method to efficiently and securely verify whether any part of data is corrupt. It is constructed by recursively computing hashes over data and their hashes (Figure 11). Its root hash summarizes the entire data. Thus, we only need to ensure the root hash's authenticity and integrity (i.e., sign it) to verify data and node hashes. For example, to verify a data block b'_1, we compute $h'_{0,1} = H(b'_1)$, $h'_{1,0} = H(h_{0,0}||h'_{0,1})$, and $h'_{root} = H(h'_{1,0}||h_{1,1})$ with leaf and internal node hashes $h_{0,0}$ and $h_{1,1}$—which have been verified in the same manner—and compare h'_{root} with signed h_{root}.

B Delta Update

The detailed procedure of the delta update is below. First, we compute an `rdiff` signature which is a structured summary of a base file (i.e., a corrupt system partition) to compute delta. In our APRON device, it takes ∼21 s to compute an `rdiff` signature over the system partition regardless of how many portions of it are corrupt. The signature size is 181 MiB without compression. Once we compress it with `gzip`, it becomes between 72 MiB (1% corruption) and 0.5 MiB (100% corruption). Next, we upload the compressed signature to the server, decompress it, and compute the delta between the signature and the valid system image. The delta computation takes 55–273 s and the size of the delta is between 91 MiB and 4.1 GiB (between 31 MiB and 1.6 GiB after compression). Both depend on how many portions of the system partition are corrupt. We only use the Azure VM for this delta computation to ignore the CPU performance difference between the two servers. Finally, we download the compressed delta—which take 1–16 s (high throughput) and 9–135 s (low throughput), decompress it, and patch the system partition with it. Patching itself takes ∼12 s regardless of the number of corrupt blocks.

zpoline: a system call hook mechanism based on binary rewriting

Kenichi Yasukata[1], Hajime Tazaki[1], Pierre-Louis Aublin[1], and Kenta Ishiguro[2]
[1]*IIJ Research Laboratory*
[2]*Hosei University*

Abstract

This paper presents zpoline, a system call hook mechanism for x86-64 CPUs. zpoline employs binary rewriting and offers seven advantages: 1) low hook overhead, 2) exhaustive hooking, 3) it does not overwrite instructions that should not be modified, 4) no kernel change and no additional kernel module are needed, 5) source code of the user-space program is not required, 6) it does not rely on specially-modified standard libraries, and 7) it can be used for system call emulation. None of previous mechanisms achieve them simultaneously.

The main challenge, this work addresses, is that it is hard to replace `syscall`/`sysenter` with `jmp`/`call` for jumping to an arbitrary hook function because `syscall` and `sysenter` are ***two-byte*** instructions, and usually more bytes are required to specify an arbitrary hook function address.

zpoline resolves this issue with a novel binary rewriting strategy and special trampoline code; in a nutshell, it replaces `syscall`/`sysenter` with a two-byte **`callq *%rax`** instruction and instantiates the trampoline code at virtual address 0. We confirmed zpoline is functional on the major UNIX-like systems: Linux, FreeBSD, NetBSD, and DragonFly BSD. Our experiments show that zpoline achieves 28.1~761.0 times lower overhead compared to existing mechanisms which ensure exhaustive hooking without overwriting instructions supposed not to be modified, and Redis and a user-space network stack bonded by zpoline experience only a 5.2% performance reduction compared to the minimum overhead case while the existing mechanisms degrade 72.3~98.8% of performance.

1 Introduction

System calls are the primary interface for user-space programs to communicate with Operating System (OS) kernels. Since user-space programs almost always go through system calls to perform important actions, system call hooks can be the vantage point to trace and change their behavior. Therefore, there are many use cases, such as tracing tools [6, 19], sandboxes [18, 25], OS emulation layers [1, 8], and binary compatibility supports of new OS subsystems [22, 29, 30, 33, 36, 37].

Motivating use case. Past studies demonstrated that user-space OS subsystems [10, 13, 17, 23, 24, 27], backed by kernel-bypass frameworks [15, 34, 38], are highly performant. In principle, system call hooks enable us to *transparently* apply user-space OS subsystems to the legacy software artifacts through the POSIX standard (as demonstrated in § 3.3), and the transparency is an important factor for the applicability of user-space OS subsystems.

Problem and related work. However, in UNIX-like systems on x86-64 CPUs, the representative platforms for server systems, there is no perfect system call hook mechanism.

1. Existing kernel supports (e.g., `ptrace` (§ 3.1.1) and Syscall User Dispatch (SUD) [20] (§ 3.1.3)) and the legacy binary rewriting technique using `int3` signaling (§ 3.1.2) **cause unacceptable performance degradation** to hook-applied user-space programs (§ 3.3).
2. Other binary rewriting mechanisms (e.g., instruction punning [7], E9Patch [9], and the technique applied in X-Containers [36]) (explained in § 2.1) and function call replacement (e.g., LD_PRELOAD (§ 3.1.4)) **cannot exhaustively hook system calls**. Thus, they cannot be used for systems requiring reliability.
3. Another type of binary rewriting technique (e.g., Detours [14]) **overwrites instructions that are supposed not to be modified** (explained in § 2.1).
4. Solutions based on specific changes to the kernel or additional kernel modules such as Dune [3], which are not merged to the mainline, substantially **diminish the portability** of applications relying on them.
5. Approaches requiring recompilation of the source code of a user-space program, typically seen in Unikernel [26] systems [5, 21], are **unusable in many cases** because users often do not have access to the source code.
6. The approach, which links application binaries with a standard library (e.g., libc) specially modified for replacing the invocations of system calls with function calls of specific OS subsystems [22, 29, 30, 33, 37], **narrows down the range of choice for applicable standard**

library implementations, moreover, **cannot hook system calls which are invoked from the outside of standard libraries**.

7. Although BSD Packet Filter (BPF) [28] and its extended version, eBPF, allow users to apply hooks to the kernel-space functions, they **cannot be used for changing and emulating the behavior of system calls without modifying the kernel source code**.

In summary, every existing system call hook mechanism has a significant downside. Due to the lack of an ideal system call hook mechanism, there have been no practical means of transparently applying user-space OS subsystems to existing user-space programs. Consequently, the applicability of user-space OS subsystems has been significantly limited, regardless of their great advantages.

Contributions. To solve this problem, we present a novel system call hook mechanism for x86-64 CPUs named *zpoline* that is free from all drawbacks mentioned above (§ 2). We demonstrate the benefits of zpoline through microbenchmarks (§ 3.2) and experiments transparently applying user-space OS subsystems to user-space programs (§ 3.3).

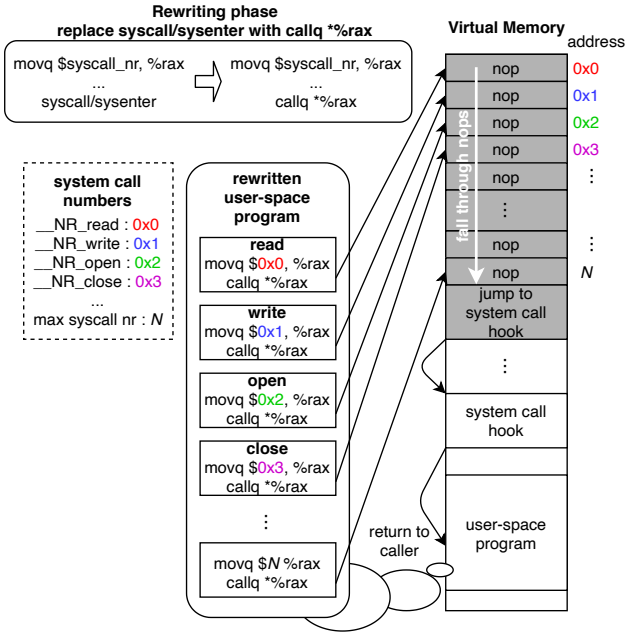


Figure 1: zpoline overview. The trampoline code is shaded.

2 zpoline

zpoline is based on binary rewriting; it replaces `syscall` and `sysenter`, which are ***two-byte*** instructions (`0x0f 0x05` and `0x0f 0x34` in opcode respectively) that trigger a system call, to jump to an arbitrary hook function address.

2.1 Challenge and Goal

The challenge of this work is that the two-byte space, originally occupied by a `syscall`/`sysenter` instruction, is too small to locate a `jmp`/`call` instruction along with an arbitrary destination address; typically, two bytes are occupied by the opcode of `jmp`/`call` and eight bytes are necessary for a 64-bit absolute address, or another possibility is one byte for a `jmp`/`call` instruction and four bytes of a 32-bit relative address. Due to this issue, existing binary rewriting techniques give up the replacement in some cases and fail to ensure exhaustive hooking [7, 9, 36], exceed the two-byte space originally occupied by `syscall`/`sysenter` to put the code bigger than two bytes while a jump to the exceeded part causes unexpected behavior [14], or take the `int3` signaling approach (§ 3.1.2) that imposes a significant overhead (§ 3.2). The goal of zpoline is to be free from these drawbacks.

2.2 Design

The overview of zpoline is shown in Figure 1.

System call and calling convention. zpoline employs the calling convention of system calls. In UNIX-like systems on x86-64 CPUs, when a user-space program executes `syscall`/`sysenter`, the context is switched into the kernel, then, a pre-configured system call handler is invoked. To request the kernel to execute a particular system call, a user-space program sets a system call number (e.g., 0 is `read`, 1 is `write`, and 2 is `open` in Linux on x86-64 CPUs) to the `rax` register before triggering a system call, and in the kernel, the system call handler executes one of the system calls according to the value of the `rax` register.

Binary rewriting. To hook system calls, zpoline replaces `syscall`/`sysenter` with **`callq *%rax`** which is represented by two bytes `0xff 0xd0` in opcode. Since the instruction sizes of `syscall`/`sysenter` and `callq *%rax` are the same two bytes, *the replacement does not break the neighbor instructions*. What `callq *%rax` does is to push the current instruction pointer (the caller's address) to the stack, and jump to the address stored in the `rax` register. Our insight is that, according to the calling convention, the `rax` register always has a system call number. Therefore, the consequence of `callq *%rax` is the jump to a virtual address between 0 and the maximum system call number which is more or less 500[1].

Trampoline code. To redirect the execution to a user-defined hook function, zpoline instantiates the trampoline code at virtual address 0; in the trampoline code, the virtual address range between 0 and the maximum system call number is filled with the single-byte `nop` instruction (`0x90`), and at the next to the last `nop` instruction, a piece of code to jump to a particular hook function is located.

[1] In Linux 5.15, the maximum system call number of the x86-64 ABI, seen in `unistd_64.h`, is 448.

Execution flow. After the trampoline code instantiation and binary rewriting are completed, the rewritten part (`callq *%rax`) will jump to one of the `nop`s in the trampoline code while pushing the caller's address on the stack. The execution slides down the subsequent `nop`s; after executing the last `nop`, it jumps to the hook function. Here, the hook function will have the same register state as the kernel-space system call handler. The return of the hook function jumps back to the caller address that is pushed on the stack by `callq *%rax`.

Security notes. We note that, like other system call hook mechanisms based on binary rewriting, zpoline itself does not offer security enhancement. On the other hand, if users wish to improve the security of zpoline-applied systems, they can employ existing mechanisms; for instance, seccomp [2] can filter the execution of kernel-space system calls triggered by a zpoline-applied user-space program, and CPU supports such as Memory Protection Keys (MPK) [16] can isolate the implementation of a hook function.

2.3 Implementation

Our current prototype focuses on Linux. The core implementation of zpoline consists of trampoline code instantiation and binary rewriting. We implement these in a shared library called `libzpoline.so` and a special loader named `zpoline_loader`; we assume a user uses either[2]. They perform the setup procedure of zpoline (§ 2.3.1) before the main function of a user-space program starts.

2.3.1 Setup Procedure

The trampoline code setup procedure first allocates memory at virtual address 0 by using the `mmap` system call[3]. Afterward, it fills the allocated memory region with the content described in § 2.2. The binary rewriting procedure initially obtains the memory mapping information from `procfs`. Then, it traverses CPU instructions on the executable memory regions, and replaces `syscall`/`sysenter` with `callq *%rax` (§ 2.2). The memory regions for the trampoline code and the code binary of the user-space program are configured to be writable during this setup phase, and they are restored to be non-writable before the setup procedure exits. After the setup completes, the main function of the user-space program starts as usual, but, all system calls are hooked by zpoline. We note that this implementation does not change the binary files of user-space programs since binary rewriting is done on the code binary loaded onto the memory.

2.3.2 Hook Function Development

zpoline users can implement an arbitrary system call hook function as part of `libzpoline.so` or `zpoline_loader`. However, there is an issue that the hook function falls into an infinite loop when it calls a function that originally executes `syscall`/`sysenter` because the replaced code (`callq *%rax`) brings the execution back to the hook function. Users encounter this issue especially when they use `libzpoline.so` because the default dynamic linker/loader automatically associates library calls used in the hook function with the libraries whose `syscall`/`sysenter` instructions are replaced with `callq *%rax`.

Use of `dlmopen`. We avoid this issue by using `dlmopen`, an extended version of `dlopen`. `dlopen` loads a library file onto the memory of a user-space process. On top of this basic feature, `dlmopen` allows users to specify a *namespace* where the library is loaded, and it conducts the association in the same namespace. Thus, `dlmopen` enables us to avoid the automatic undesired association by loading the hook function in a new namespace. To use `dlmopen`, we assume a zpoline user builds the core of the hook implementation as an independent shared library. During the setup phase, `libzpoline.so` loads the library using `dlmopen`, and obtains the pointer to the core implementation of the hook function by using `dlsym`. The hook function, implemented in `libzpoline.so`, calls it through the obtained pointer.

2.3.3 NULL Access Termination

Typically, a memory access to virtual address 0, namely the NULL pointer access, causes a page fault because of the lack of physical memory mapping at virtual address 0, and it results in the termination of the user-space program. The NULL access termination is important for stopping buggy programs. On the other hand, the use of virtual address 0 in zpoline brings about the issue that the NULL access of a user-space program does not cause a fault. To cope with this issue, zpoline employs a set of techniques.

Terminate NULL read and write. To terminate NULL read and write, zpoline configures the trampoline code to be the eXecute-Only Memory (XOM)[4]; a user-space program, that attempted a read/write access to XOM, will be terminated by the kernel because of a fault.

Terminate NULL execution. To trap unintentional NULL execution, zpoline collects the virtual addresses of `syscall`/`sysenter` which are replaced during the setup phase of zpoline (§ 2.3.1), and it checks, at the entry point

[2] `libzpoline.so` assumes to be loaded through LD_PRELOAD and used when the application binary is dynamically linked. LD_PRELOAD allows `libzpoline.so` to run the setup procedure before the main function of the user-space program starts. `zpoline_loader` is complementary and assumes to be used when the application binary is statically linked and the LD_PRELOAD feature does not work.

[3] In Linux, by default, the memory mapping to virtual address 0 is only allowed for the root user, but it can be permitted for all non-root users by setting 0 to `/proc/sys/vm/mmap_min_addr` (confirmed in Linux 5.15).

[4] In Linux running on a CPU supporting the Memory Protection Keys (MPK) [16] feature, the `mprotect` system call configures XOM when only `PROT_EXEC` is set in the access flag; if MPK is not supported by the CPU, `mprotect` does not configure XOM.

of the hook function, if the caller of the hook function is one of the replaced virtual addresses or not. If it is not, zpoline terminates the user-space program because it is not the jump from the replaced `callq *%rax`, meaning an unintentional jump to NULL. To maintain the replaced virtual addresses while achieving a low-overhead NULL execution check, we use a bitmap that covers the entire 256 TB (48-bit) virtual address range that is typical in x86-64 CPUs. This bitmap allows us to conduct the NULL execution check with a few bit operations, and this cost is evaluated in § 3.2. The bitmap occupies 32 TB of virtual address space, however, its physical memory consumption is substantially smaller because the virtual address pages, whose all bits are clear, do not need to have underlying physical memory pages[5]. We note that if a user prefers to avoid occupying 32 TB of virtual address for the bitmap, we can alternatively use a hash table at the cost of the higher overhead of the NULL execution check.

2.4 Limitations

Here, we discuss the limitations of zpoline.

`syscall`/`sysenter` loaded at runtime. The current prototype of zpoline cannot hook `syscall`/`sysenter` loaded or crafted after the completion of the setup (§ 2.3.1). We can resolve this issue by borrowing the idea of online binary rewriting presented in the X-Containers [36] work that traps an invocation of a system call and rewrites, on the fly, the `syscall`/`sysenter` instruction that triggered the system call.

vDSO (virtual dynamic shared object). Kernels provide user-space programs with several system calls by directly exposing the code for them through vDSO. Like other system call hook mechanisms, zpoline cannot hook vDSO-based system calls by default; however, we can enable zpoline to hook them by disabling vDSO[6].

Unusable virtual address 0. zpoline is not applicable if memory at virtual address 0 is unusable; for instance, the virtual address 0 is already used for other purposes, or the kernel does not allow the mapping at virtual address 0.

Other OSes. We confirmed that zpoline is functional on FreeBSD 13.0, NetBSD 9.2, and DragonFly BSD 6.0[7]. We could not use zpoline on OpenBSD 7.0 because the minimum mappable virtual address is hard-coded as the page size. In Windows, `VirtualAlloc` is conceptually equivalent to `mmap`. On Windows 10, `VirtualAlloc` fails when the specified virtual address is lower than 0x10000, therefore, we could not apply zpoline. But, Windows offers a compatibility layer for Linux called Windows Subsystem for Linux (WSL). We confirmed that zpoline works on WSL2 while it did not on WSL1 whose `mmap` to virtual address 0 returns successfully but actually does not conduct the memory mapping. On macOS, the virtual address 0 of a user-space program is used by a special segment named `__PAGEZERO`, therefore, we could not apply zpoline on macOS.

Other CPU architectures. zpoline is not compatible with CPU architectures which assume the instructions to be aligned by architecture-specific sizes on the memory and consider a jump to an unaligned virtual address as an invalid operation (e.g., ARM); this is because, when zpoline is applied, the execution can jump to an unaligned virtual address between 0 and the maximum system call number (§ 2.2)[8]. However, we believe zpoline is applicable to a large number of servers because x86-64 CPUs are very popular.

3 Evaluation

This section evaluates zpoline through a comparison with existing hook mechanisms (§ 3.1). Particularly, we quantify the hook overhead of zpoline (§ 3.2) and the performance penalty experienced by application programs and user-space OS subsystems bonded by zpoline (§ 3.3).

Experiment setup. For the experiments, we use two machines; each has two 16-core Intel Xeon Gold 6326 CPUs clocked at 2.90 GHz and 128 GB of DRAM. The two machines are directly connected via Mellanox ConnectX-5 100 Gbps NICs. In the experiments in § 3.3, we use one of the two as the *server machine*, and the other as the *client machine*. Both machines run Linux 5.15.

3.1 Comparison

We compare zpoline with `ptrace` (§ 3.1.1), `int3` signaling (§ 3.1.2), SUD (§ 3.1.3), and LD_PRELOAD (§ 3.1.4). Here, we describe the mechanisms and properties of them.

3.1.1 `ptrace`

UNIX(-like) OSes offer the `ptrace` system call that enables a *tracer* process to hook system calls attempted by a *tracee* process. Since `ptrace` is a kernel feature, it can hook system calls exhaustively. However, its hook overhead is enormous due to the context switch between the tracer and tracee; the tracer sleeps while the tracee is running, and the tracee sleeps during the tracer runs its hook function. Therefore, at every system call invocation, the tracee experiences a long latency that includes the wake-up time of the tracer, the execution time of the hook function, and the wake-up time of the tracee. This latency results in *significant performance degradation of the user-space program running on the tracee.*

[5] In many cases, most of `syscall`/`sysenter` instructions come from libc and ld.so. We found the bitmap uses 22 and 5 physical 4 KB pages to maintain 544 and 50 of `syscall`/`sysenter` in libc and ld.so respectively (glibc-2.35).

[6] Linux disables vDSO when the kernel boot option specifies `vdso=0`.

[7] In FreeBSD and NetBSD, users can use `sysctl` to permit memory mapping at virtual address 0. DragonFly BSD allows it by default.

[8] Besides the issue of the instruction alignment, binary rewriting techniques need to pay attention to architecture-specific factors; for example, on ARM CPUs, the simple replacement from `SVC` to `BL` overwrites/breaks the return address saved in a specific register [31].

3.1.2 `int3` Signaling

`int3` is a one-byte instruction (`0xcc`) that invokes a software interrupt. On Linux, the kernel handles it and raises `SIGTRAP` to the user-space process that executed `int3`. The `int3` signaling technique exploits this behavior to hook system calls; it replaces `syscall`/`sysenter` with `int3` and employs the signal handler for `SIGTRAP` as the hook function. Since `int3` is one byte, it can replace an arbitrary instruction without breaking the neighbor instructions. This technique is traditionally used in debuggers to implement breakpoints. However, *signal handling incurs a large overhead* because it involves context manipulation by the kernel.

3.1.3 Syscall User Dispatch (SUD)

Syscall User Dispatch (SUD) [20] was added in Linux 5.11, and it offers a way to redirect system calls to arbitrary user-space code. For the SUD feature, the kernel implements a hook point at the entry point of system calls. A user-space process can activate SUD via the `prctl` interface. When SUD is activated, the hook point raises `SIGSYS` to the user-space process. This mechanism allows a user-space program to leverage the `SIGSYS` signal handler as the system call hook. However, similarly to the `int3` signaling technique, SUD *imposes a significant performance penalty on the user-space program due to the overhead of the signal handling.*

3.1.4 Function Call Replacement by LD_PRELOAD

The dynamic linker/loader (`ld.so`) offers the LD_PRELOAD feature that allows users to specify shared objects to be loaded before the main part of a program starts, and it can be used for selectively overriding function calls implemented in other shared objects. Users can employ this mechanism to replace the system call wrapper functions, which are typically implemented in standard libraries, with arbitrary function calls. The performance penalty of LD_PRELOAD is very small because the hooks are applied through function pointer replacement.

A function call hook is not a system call hook. However, precisely, the function call replacement for a system call wrapper function is not the hook for a system call; in the first place, the `syscall` and `sysenter` instructions are not directly associated with any function calls, and LD_PRELOAD cannot hook a `syscall`/`sysenter` instruction which does not have a dedicated and exported wrapper function.

The case where LD_PRELOAD fails to hook. glibc [11] is a representative example where LD_PRELOAD cannot apply system call hooks exhaustively. In many cases, glibc does not use the well-known system call wrapper functions to invoke system calls; instead, glibc directly embeds `syscall`/`sysenter` in its internal functions which are marked as invisible from the outside of glibc, and LD_PRELOAD cannot apply hooks to `syscall`/`sysenter` instructions wrapped by such internal function calls.

Mechanism	Time [ns]
`ptrace`	31201
`int3` signaling	1342
SUD	1156
zpoline	41
zpoline (no NULL execution check (§ 2.3.3))	40
LD_PRELOAD	6

Table 1: The overhead for hooking a system call.

Potential but impractical approach. Although it is possible for users to apply hooks using LD_PRELOAD by entirely replacing library calls that contain `syscall`/`sysenter` instructions, this approach does not scale because users must give up the use of the original library call implementations; in other words, they need to reimplement the equivalent functionalities by themselves, however, it is not realistic to reimplement large part of glibc. Moreover, this reimplementation approach cannot be applied if, unlike glibc, the source code of a shared library file is not available.

Limitation of LD_PRELOAD. In short, *LD_PRELOAD cannot exhaustively hook system calls*, thus, is not an appropriate option to apply user-space OS subsystems to existing user-space programs; for instance, a file descriptor, which is opened by a user-space OS subsystem, will be passed to a kernel-space OS subsystem if a system call is not properly hooked, and it leads to unexpected behavior of the system.

Similarity to binary rewriting techniques. We note that, in our experiments, the cases of other binary rewriting techniques [7, 9, 14, 36] are represented by the LD_PRELOAD case because they share the same characteristics: their performance overhead is very small, however, as described in § 2.1, they cannot hook system calls exhaustively.

3.2 System Call Hook Overhead

We quantify the system call hook overhead by measuring the time to hook `getpid`, one of the simplest system calls. Our primary interest here is the hook overhead itself; to avoid the overhead of the kernel-crossing system call, we use a hook function that returns a dummy value without actually executing the `getpid` system call. Table 1 shows the results. First, the overhead of LD_PRELOAD is negligible as expected (§ 3.1.4). The overhead of zpoline is 6.8 times higher than LD_PRELOAD, and this is primarily due to the `nop`s in the trampoline code (§ 2.2). The cost of the NULL execution check (§ 2.3.3) is 1 ns out of 41 ns. zpoline is 761.0, 32.7, and 28.1 times lighter than `ptrace`, `int3` signaling, and SUD respectively. The major overheads of `int3` signaling (§ 3.1.2) and SUD (§ 3.1.3) derive from the signal handling for `SIGTRAP` and `SIGSYS`. `ptrace` exhibits the biggest overhead due to the cost of scheduling between the tracer and tracee processes (§ 3.1.1).

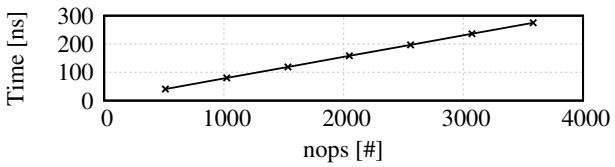


Figure 2: The overhead to hook a system call depending on the number of `nop`s at the beginning of the trampoline code.

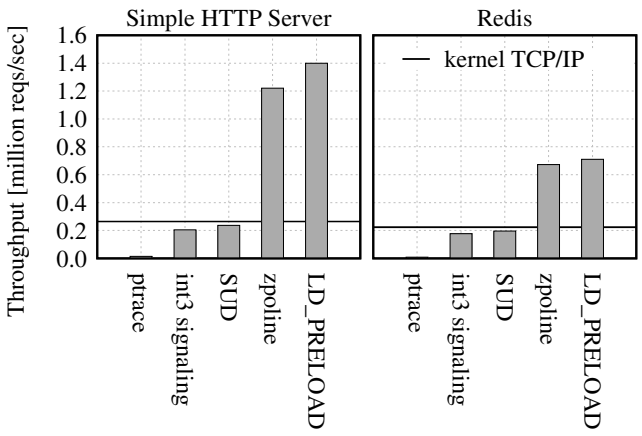


Figure 3: Performance of applications running on lwIP and DPDK with different system call hook mechanisms. For reference, the throughput of the Linux kernel TCP/IP stack is shown by the non-dotted horizontal line.

nop overhead. In zpoline, `nop`s in the trampoline code increase the hook overhead. The number of `nop`s depends on the number of system calls implemented by the kernel (§ 2.2). To see how the overhead grows, we run the same `getpid` test above while changing the number of `nop` instructions in the trampoline code. Figure 2 shows that the overhead linearly increases according to the number of `nop`s. However, in the first place, the kernel development communities are very wary to add system calls. Therefore, we believe the `nop` overhead of zpoline will not increase drastically in the future. Moreover, although 3.5 K `nop`s are located, the overhead of zpoline is still 113.4, 4.8, and 4.2 times lower than that of `ptrace`, `int3` signaling, and SUD respectively.

3.3 User-space OS Subsystem Performance

This section evaluates how zpoline affects the performance of application programs backed by user-space OS subsystems; we employ zpoline and the existing hook mechanisms described in § 3.1 to *transparently* apply a portable TCP/IP stack, lwIP [10], backed by Data Plane Development Kit (DPDK) [15], to a simple HTTP server and Redis [35]. Normally, kernel-bypassing lwIP achieves higher networking performance than the kernel TCP/IP stack of Linux [4, 32]; for reference, we run the same benchmarks using the kernel TCP/IP stack of Linux and report its performance by non-dotted horizontal lines in Figure 3. We note that the simple HTTP server and Redis are chosen for the experiments because LD_PRELOAD could apply hooks to them, and as explained in § 3.1.4, LD_PRELOAD can fail to hook system calls in other systems.

Simple HTTP server. Commonly, a server program triggers network-relevant system calls more frequently when its application logic gets lighter because it can serve a lot of requests in a short time. To stress the hook mechanisms with lightweight application logic, we made a simple HTTP server that replies a static 64-byte content; we run it on the server machine. As the benchmark client, we run wrk [12] on the client machine; it sends requests through 32 persistent concurrent connections. The results are shown in Figure 3 (left). First, the LD_PRELOAD result represents the minimum overhead case (§ 3.2), and it demonstrates the potential of lwIP on DPDK, which is 5.2 times faster than the Linux kernel TCP/IP stack whose throughput is shown by the non-dotted horizontal line in Figure 3 (left). Comparison with the LD_PRELOAD case sheds light on the overhead of each hook mechanism. The percentages of performance reduction in `ptrace`, `int3` signaling, and SUD compared to LD_PRELOAD are 98.9%, 85.3%, and 83.0% respectively. Contrarily, zpoline causes only 12.7% of performance reduction. These results are explained by the hook overheads shown in Table 1.

Redis. We evaluate how a real-world application performs on zpoline. For benchmarking, we use Redis [35], a widely used key-value store; we run a Redis server process on the server machine. As the benchmark client, we use `redis-benchmark`, which is distributed as part of the Redis source, on the client machine; we run the GET 100% workload so that the Redis server will spend most of its time on networking operations rather than disk operations. Requests are sent over 32 persistent concurrent connections. Figure 3 (right) shows a similar trend to the simple HTTP server experiment, and the overall results reflect the overheads shown in Table 1. Compared to LD_PRELOAD, the throughput results of `ptrace`, `int3` signaling, and SUD are 98.8%, 75.0%, and 72.3% lower respectively. In contrast, zpoline imposes only 5.2% of throughput reduction.

4 Conclusion

This paper has presented zpoline, a system call hook mechanism for x86-64 CPUs, that can exhaustively hook system calls at a low overhead without overwriting instructions that are supposed not to be modified. zpoline is a practical means of transparently applying user-space OS subsystems to existing user-space programs and contributes to the applicability of user-space OS subsystems.

Acknowledgments

We are grateful to anonymous USENIX ATC 2023 and 2022 reviewers and Pierre Olivier, our shepherd at USENIX ATC 2023, for their insightful comments.

References

[1] Bob Amstadt and Eric Youngdale. Wine. https://www.winehq.org/, 1993.

[2] Andrea Arcangeli. seccomp. https://man7.org/linux/man-pages/man2/seccomp.2.html, 2005.

[3] 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.

[4] 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.

[5] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*, pages 250–257. IEEE, 2015.

[6] Juan Cespedes. ltrace. https://ltrace.org/, 1997.

[7] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 320–332, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Jeff Dike. User-mode linux. In *5th Annual Linux Showcase & Conference (ALS 01)*, 2001.

[9] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Adam Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2(77), 2001.

[11] Free Software Foundation. The GNU C Library (glibc). https://www.gnu.org/software/libc/, 1988.

[12] Will Glozer. wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk, 2012.

[13] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. Rekindling network protocol innovation with user-level stacks. *SIGCOMM Comput. Commun. Rev.*, 44(2):52–58, apr 2014.

[14] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, WINSYM'99, page 14, USA, 1999. USENIX Association.

[15] Intel. Data Plane Development Kit. https://www.dpdk.org/, 2010.

[16] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals, 2023.

[17] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[18] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 139–144, San Jose, CA, June 2013. USENIX Association.

[19] Paul Kranenburg. strace. https://strace.io/, 1991.

[20] Gabriel Krisman Bertazi. Syscall User Dispatch. https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html, 2021.

[21] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[23] 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*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Google LLC. gVisor. https://gvisor.dev/, 2018.

[26] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 175–186, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[29] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 116–132, New York, NY, USA, 2013. Association for Computing Machinery.

[30] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 59–73, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Pierre Olivier, Hugo Lefeuvre, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A syscall-level binary-compatible unikernel. *IEEE Transactions on Computers*, 71(9):2116–2127, 2022.

[32] 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.

[33] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. Unikernel linux (ukl). In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 590–605, New York, NY, USA, 2023. Association for Computing Machinery.

[34] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[35] Salvatore Sanfilippo. Redis - Remote Dictionary Server. https://redis.io/, 2009.

[36] 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, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[38] 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.

Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks

Won Wook Song
Seoul National University

Taegeon Um
Samsung Research

Sameh Elnikety
Microsoft Research

Myeongjae Jeon*
UNIST

Byung-Gon Chun*
Seoul National University and FriendliAI

Abstract

Streaming workloads deal with data that is generated in real-time. This data is often unpredictable and changes rapidly in volume. To deal with these fluctuations, current systems aim to dynamically scale in and out, redistribute, and migrate computing tasks across a cluster of machines. While many prior works have focused on reducing the overhead of system reconfiguration and state migration on pre-allocated cluster resources, these approaches still face significant challenges in meeting latency SLOs at low operational costs, especially upon facing unpredictable bursty loads.

In this paper, we propose Sponge, a new stream processing system that enables fast reactive scaling of long-running stream queries by leveraging serverless framework (SF) instances. Sponge absorbs sudden, unpredictable increases in input loads from existing VMs with low latency and cost by taking advantage of the fact that SF instances can be initiated quickly, in just a few hundred milliseconds. Sponge efficiently tracks a small number of metrics to quickly detect bursty loads and make fast scaling decisions based on these metrics. Moreover, by incorporating optimization logic at compile-time and triggering fast data redirection and partial-state merging mechanisms at runtime, Sponge avoids optimization and state migration overheads during runtime while efficiently offloading bursty loads from existing VMs to new SF instances. Our evaluation on AWS EC2 and Lambda using the NEXMark benchmark shows that Sponge promptly reacts to bursty input loads, reducing 99^{th}-percentile tail latencies by 88% on average compared to other stream query scaling methods on VMs. Sponge also reduces cost by 83% compared to methods that over-provision VMs to handle unpredictable bursty loads.

1 Introduction

Stream queries continuously process real-time data to extract insights and make business-critical decisions, such as analyzing real-time logs to extract statistics, detect anomalies, and provide notifications [2, 6, 29, 48, 52]. Latency is an essential service level objective (SLO) in these streaming workloads, as faster up-to-date results mean more value. Stream systems are expected to run 24/7 while meeting their SLOs [53].

Meanwhile, stream systems regularly face significant challenges due to sudden, unpredictable bursts of input loads caused by random events, e.g., influencer tweets, breaking news, and natural disasters [46, 47]. These bursts can abruptly increase the input load by more than 10× in just a few seconds [11, 17, 26, 37, 56]. If stream processing systems do not quickly acquire additional computing resources that can handle the bursty loads and do not promptly redistribute the load to the newly allocated computing resources, events will soon pile up on the existing resources, leading to cascading impacts on query latencies that can have fatal consequences such as reduced user satisfaction and revenues [48].

One approach to quickly acquiring additional computing resources is to over-provision resources. Existing work such as Rhino [18], Megaphone [25], and Chronostream [55] builds efficient stream load redistribution mechanisms by harnessing over-provisioned resources to minimize latency spikes on load bursts. For instance, Megaphone [25] smoothly migrates stream query loads to extra resources during stable load in preparation for load spikes. However, over-provisioning solutions can be *costly* and inefficient, as a significant amount of resources will remain idle for most of the time.

Cloud services can reduce operational costs by offering on-demand resource allocations. Existing scaling approaches for on-demand resources dynamically migrate stream operator instances, in units of parallel *tasks*, to the allocated on-demand virtual machines (VMs). They redistribute the tasks and their states, which are key-value pairs of aggregated intermediate results [7, 15, 16, 19, 36, 49]. However, migrating tasks and their states incurs extra overheads (e.g., (de)serialization), which increase proportionally to the state size (e.g., a large number of key-value pairs), and can violate low latency SLOs. Moreover, using VMs, which are popular on-demand cloud resources, can further exacerbate latency spikes due to the considerable launch delay of VM instances which can take

* Corresponding authors.

dozens of seconds (i.e., 25-30 secs) with conventional cloud providers [21, 31, 44].

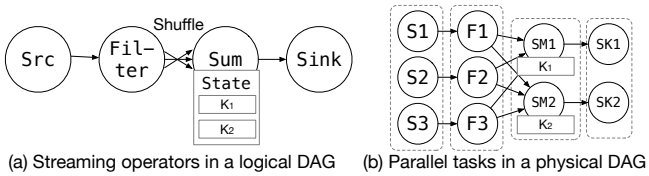
In this paper, we design Sponge, a new stream processing system that requires low operational costs and keeps low latency upon sudden bursty loads. Sponge is designed with the following three design principles:

Combining two heterogeneous cloud resources to have the best of both worlds: Sponge harnesses two heterogeneous cloud resources: VMs and serverless function (SF) instances. Serverless solutions provided by conventional cloud providers [11, 17, 26, 37, 56] only take hundreds of milliseconds (i.e., 300-750 ms) to launch and prepare and are designed to achieve high scalability, while the operational costs are much higher than those of VMs. Therefore, to achieve low latency and low operational costs, Sponge uses VMs for processing stable streaming loads for longer periods of time, while quickly invoking SF instances and using them for short periods of time to handle bursty loads. If the bursty input loads persist, we may consider launching new VM instances to permanently offload the tasks with existing state migration techniques [16, 18, 19, 23, 25, 28, 36, 45, 49, 55]. In such cases, on-demand SF instances can be used to accomplish system SLOs by hiding the launch overhead during the preparation of the new VM instances.

Keeping tasks with high migration overheads on VMs, while quickly redirecting data to SFs: When VMs process streaming data with stable loads over long periods of time, the states of stream tasks are materialized, and the state size may increase on the existing VMs. To avoid the state migration overheads from VMs to SFs, Sponge incorporates the *redirect-and-merge* mechanism: Sponge immediately redirects the increased load to SFs, which are imminent to offload, so that each SF instance can build small partial states and periodically send them back to the VMs to merge with the original states. This approach allows Sponge to promptly exploit fast-launching SF instances and bypass the prohibition of direct network communication between SF instances. For quick data redirection, Sponge exploits SF properties to prevent cold start latencies and pre-initiates copies of VM tasks on SFs to keep its runtime, process, and pre-initiated tasks readily available on time.

***Fast reactive scaling*:** On top of the fast resource scaling mechanisms on SF instances, Sponge identifies bottleneck tasks *reactively* and makes precise decisions on which part of the query to offload and how much of the compute resources to request. At runtime, Sponge continuously monitors the CPU usage, the major resource constraint of task execution, to quickly react to the changing input loads. Our offloading policy determines the fraction of input loads to offload based on excess events accumulated in the input queue and accounts for the optimal time to recover from load increases to meet the SLOs for a given query.

Sponge is built atop Apache Nemo [51, 57] with about 10K lines of code. We evaluate Sponge on EC2 instances (5× r5.xlarge) and AWS Lambda instances (up to 200 Lambda instances of 1,769MB memory with one full CPU core) with NEXMark [42], a popular benchmark for stream processing. The effectiveness of our optimizations varies according to the characteristics of queries (e.g., dataflow pattern, # of tasks, and state size). Our evaluations show that Sponge exhibits similar performance to costly over-provisioned approaches, and reduces input event 99^{th}-percentile tail latencies by 88% on average compared to scaling queries on VMs and by 70% compared to scaling on SFs without our techniques.



(a) Streaming operators in a logical DAG (b) Parallel tasks in a physical DAG

Figure 1: (a) Logical DAG of four operators including a stateful `Sum` operator with two key groups. (b) The corresponding physical DAG with parallel tasks.

2 Background

In this section, we describe the resource demand characteristics of stream processing and different on-demand resource provisioning methods provided by current cloud services.

2.1 Stream Processing

Execution model. A stream processing query processes an unbounded number of timestamped events to derive specific results (e.g., top K, statistics) on every temporal window. The execution of the query is generally expressed as a directed acyclic graph (DAG) of operators and data dependencies. As shown in Fig. 1, a vertex represents a stream operator that transforms input events and emits output events, and an edge represents how data flows between its adjacent operators. Popular stream engines like Flink [15], Spark Streaming [7], and Beam [12] aid users with high-level languages (e.g., declarative language) to facilitate query expressions. To provision compute resources over stream operators in response to the input data rate, the stream engine generates an optimized physical DAG (Fig. 1(b)) after translating a user query into a logical DAG (Fig. 1(a)). In a physical DAG, each logical operator is expanded into n parallel tasks, $p_0, ..., p_{n-1}$, where each task processes a disjoint data partition.

Streaming operators and resource demands. A stream operator is either stateless or stateful. Stateless operators, such as `map` and `filter`, are typically used to compute individual events or drop unnecessary events or fields by applying predicates. Due to their simplicity, stateless operators can be pipelined together within a single node to leverage data locality and reduce network overheads. On the other hand, stateful operators, such as `groupByKey` and `join`, perform data

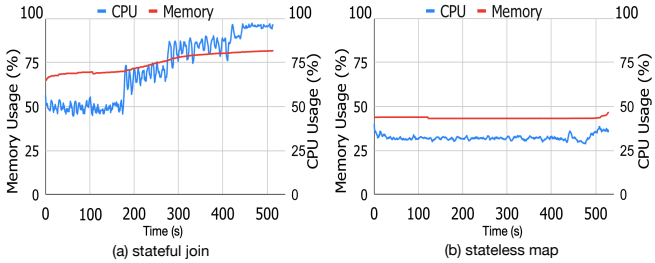


Figure 2: CPU and memory usage patterns for (a) stateful windowed `join` and (b) stateless `map` operators upon processing a fixed input rate of 80K events/s on identical 4 vCore nodes. The CPU and memory usage of the stateful operator increase until the window is full.

grouping within a window boundary to organize unbounded streaming events into disjoint groups based on timestamps and aggregation keys, requiring computationally extensive key lookups. Thus, most streaming engines apply parallelism specifically to stateful operators such that a single stateful task p_i processes events that only belong to a non-overlapping key partition group K_i out of the entire key space $K = \cup_{i=0}^{n-1} K_i$.

Stateful operators are often the major source of system bottlenecks [38, 50]. In particular, since each parallel stateful task is assigned to a key partition group, it incurs shuffle communication for the events in its key group that are collected from the preceding (upstream) operators. Shuffle communication often requires the data to travel across different nodes, requiring data serialization and deserialization on top of the computation performed for the key lookups. As a result, as shown in Fig. 2, it is prevalent to provision more CPUs to execute stateful operators rather than stateless operators [28, 54].

2.2 On-Demand Resource Provisioning

Several real-world stream analytics systems report high temporal variability in the event count of data streams, even across one-minute time windows [34, 36, 43, 48]. This means that stream processing may need to frequently adjust resource provisioning and query execution plans in response to changes in input loads. Upon facing increased input loads, the system needs to allocate more resources to avoid operators being congested and maintain stable query latency.

Cloud providers offer primarily two options for on-demand resource allocation: virtual machines (VM) and serverless functions (SF). We compare three representative characteristics between these two options in more detail.

Resource size. VMs are machine-isolated by bare-metal hypervisors, whereas SFs are process-isolated by OSes. Therefore, SFs are much more flexible in allocating resources. Cloud providers typically provide VMs in chunks of a predefined, fixed amount of resources (e.g., r5.xlarge with 4 vCores and 32GB memory). In contrast, SFs are allocated based on a specified memory size. For the memory size, cloud providers assign a certain number of CPU power (e.g., vCores)

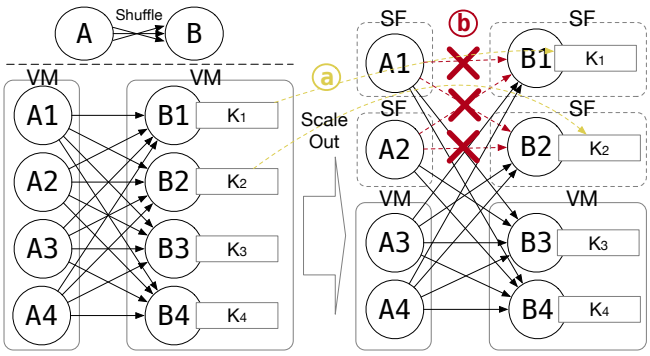


Figure 3: While scaling out on SF instances, the system must be aware that ⓐ task state migration overheads lead to latency spikes, and ⓑ direct data communication among adjacent tasks is prohibited between SF instances.

guided by their pricing model [8]. We observe that network bandwidth per SF instance is about 100Mbps and concurrently using multiple SFs can increase the bandwidth up to GBs of effective bandwidth, which VMs already support, providing enough capacities to handle most streaming workloads.

Start-up time. VM instances take a significant amount of time to launch and to prepare the runtime stack for query workload as they virtualize resources using bare-metal hypervisors. We observe that provisioning a new VM instance in major cloud service providers, like AWS, Azure, and GCP, mostly has a latency of over 25 seconds. On the contrary, SF instances provided by these cloud vendors take only 300-750 ms to launch and be ready to run because SF instances share runtimes and resources at the OS level.

Usage cost. SF instances are much more expensive to use than VM instances, e.g., 4× more expensive when running a 1GB SF instance with AWS Lambda (with $<$ 1 vCPU) compared to a t2.micro EC2 instance, which is equipped with 1 vCPU and 1GB RAM. However, temporarily using SF instances primarily for frequent short-lived bursty loads that constitute only a small fraction of time throughout the day [26] does not significantly increase the operational cost (§ 6.5).

3 Challenges

Based on these observations, we propose to use a combination of VMs and SFs to have the best of both worlds. To achieve low latency and cost, we use cheap and stable VMs for handling continuous loads for long periods of time, and costly and reactive SFs for bursty loads during short periods of time. In this section, we describe several challenges in scaling streaming loads from VMs to on-demand SF instances.

C1. Migration with large operator states. For stream scaling in the cloud, existing approaches trigger resource adaptation primarily by re-scaling operators (i.e., increasing or decreasing parallelism) and migrating the bottlenecked tasks to the instances with available resources (i.e., load redistribution) [7, 15, 16, 19, 28, 36, 49]. Thus, even if we can set

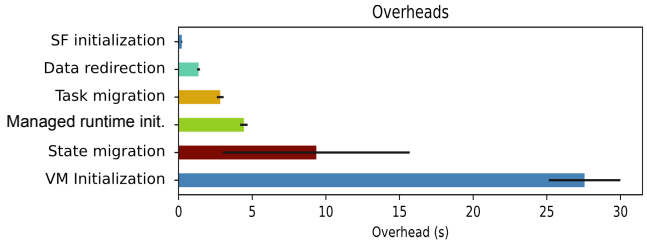


Figure 4: A comparison of the overheads of different steps of workload scaling on stream processing systems in the cloud. The VM, SF, and managed runtime initialization overheads are averaged across all instances, and the data redirection and task/state migration overheads are averaged across all single-scaling operations for the 5× input load experiment in § 6. Error bars indicate the 95% confidence interval.

up SF instances quickly, the task state migration overhead is inevitable with existing systems, as shown by Fig. 3ⓐ, and paradoxically often inflicts damage to system performance.

Fig. 4 illustrates the various overheads that occur during a single workload scaling for the queries evaluated in § 6. As shown in this figure, the task migration and reconfiguration require a few extra seconds (3-4 seconds) to resume the work after the migration. Also, the state migration takes several seconds (e.g., from 4 to 17 seconds) depending on the state size because of the (de)serialization overheads of states. These task and state migration overheads lead to increased query latency due to the delay in receiving events from upstream tasks. The system that aims to meet low-latency SLOs must correctly and rapidly carry out task offloading to SFs. In particular, some use cases are expected to generate outputs even in order of seconds or less, without query accuracy loss [36, 48].

C2. Indirect data communication between SF instances. As SFs are designed to be provisional and temporary, cloud vendors usually prohibit running a server process that can accept inbound network connections on an SF instance. Hence, direct data communication across SF instances is prohibited. This prevents neighboring stream operators (parent and child operators) from being offloaded to SFs simultaneously, as these operators require direct shuffle data transfers to group data by its key partitions, as shown in Fig. 3ⓑ.

Therefore, we can choose to migrate only certain tasks to SFs (e.g., either operator A or B in Fig. 3), but this eventually leads the bursty input load to end up on VMs on the adjacent operators and fails to alleviate latencies. Alternatively, we can offload all the tasks involved in the shuffle communication on a large SF instance. However, this forces parallel tasks to be located on a single SF instance, which can lead to network pressure while leaving VMs idle. Consequently, it is essential to design the system to be able to offload adjacent operators together to SFs while bypassing the prohibited direct communication across SF instances.

C3. Quick decision making and scaling. With frequent unpredictable changes in input events, offloading decisions must be made quickly at runtime. Stream systems often detect symptoms of bottlenecks from system metrics and decide on whether and how much to scale. However, existing approaches can be too slow, as they require multiple iterations of optimization that scale bottleneck operators one after another [19]. Other work prevents such iterations by providing a global optimum after collecting all metrics from all executors to redistribute tasks [28]. While these approaches effectively find the target throughput and may be suitable for throughput-oriented workloads, they only work in intervals of multiple 10s of seconds and may not be suitable for latency-oriented workloads. For a stream system operating with diverse intervals and window sizes, it is important to have a uniformly fast and effective optimization level to prevent window outputs from being delivered too late.

4 Sponge Design

In this section, we describe the key pillars of our system design and explain the details by illustrating our graph rewriting algorithm, dynamic offloading policy, and the mechanisms that prevent cold start latencies and enable system correctness.

4.1 Design Overview

Latency spikes occur when the input rate r_i exceeds the maximum throughput m_i on a particular task p_i. When this happens, data starts to accumulate on the event queue, along with the operator state in memory, leading to high CPU usage and memory pressure. In a cloud environment, the maximum throughput m_i often depends on the CPU capacity allocated to the task, regardless of the operator type. This is because most cloud providers are equipped with GBs of network bandwidths, and memory pressure starts to increase when the CPU becomes saturated, and the input data builds up in the event queue with $r_i > m_i$. Therefore, our goal is to primarily focus on relieving CPU pressure. To achieve this, we design a system that accurately estimates the amount of additional resources needed and provides fast mechanisms for offloading CPU computation from VMs to SFs through two design principles: *redirect-and-merge* and *fast reactive scaling*.

Redirect-and-merge. Sponge is designed to rapidly forward increased input load to available resources in SF instances. To ensure speed, we bypass expensive query optimizations during runtime by performing DAG optimizations during compile time, i.e., when the application is launched (Fig. 5ⓐ). During compile time, there are no concerns yet about runtime synchronization and progress, so it only takes about 200ms upon workload initialization to perform the DAG optimizations. After the optimization, Sponge scheduler places tasks on appropriate executors (Fig. 5ⓑ). This allows Sponge to focus on which operators and how much of their data volume

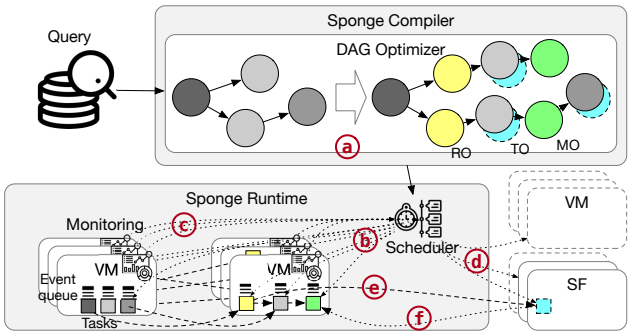


Figure 5: Sponge architecture.

to offload to SFs based on monitoring CPU usage (Fig. 5ⓒ) without having to relaunch queries at runtime.

While stateful operators are our primary focus as initial bottlenecks, any operator can become a subsequent bottleneck. Thus, we enable offloading for any operator, regardless of its type and statefulness. We design transient operators (TOs) so that operator logic can be prepared on SF instances to receive events immediately after detecting an increase in the input load and CPU usage on VMs (Fig. 5ⓓ). We also enable offloading to be activated at any time with high efficiency and responsiveness. To meet these requirements, we introduce a set of new proxy operators: router operators (ROs) and merge operators (MOs). ROs supervise the data communication to downstream VM and SF instances, in order to enable the system to rapidly and elastically *forward* data from any existing operators to the designated instances (Fig. 5ⓔ). To minimize state migration overhead, which is a major bottleneck in task migration [18, 23, 25, 55], the states, exclusively for the offloaded input load, are maintained separately on SFs. MOs enable the system to later *merge* the corresponding states of offloaded workload created on SFs with the states on the original VMs for any stateful operators (Fig. 5ⓕ). This way, the offloading overhead for both stateful and stateless operators is substantially reduced, as we only have to offload the computational logic, and not the states. The proxy operators are inactive during non-scaling periods to avoid extra costs and are only activated when needed.

Fast reactive scaling. With the principle above, we provide a fast reactive approach that prevents inaccurate predictions on resource provisioning by monitoring local performance metrics within the executors. Bottlenecks often occur individually on VMs, so it is sufficient to mitigate them *locally* within each VM. As briefly mentioned, relieving CPU pressure when the input rate r_i is greater than the operator throughput m_i is key to reducing CPU and memory strain in stream processing systems. Sponge has low monitoring overhead, with less than *10ms per observation*. Based on input rate and CPU usage observations, Sponge estimates the amount of CPU resources needed to increase operator throughput and meet our SLOs under increased input loads.

Algorithm 1: DAG rewriting for operator insertion.

```
1  Function OperatorInsertion(dag)
2      for vertex, inedges in dag.topological_sort() do
3          t_op = TransientOp.for(vertex)
4          for inedge in inedges do
5              if inedge.comm != 1to1 then
6                  r_op = RouterOp.create()
7                  dag.remove_edge(inedge)
8                  e1 = {inedge.src→r_op, inedge.comm}
9                  e2 = {r_op→vertex, 1to1}
                   // connect transient operators
10                 e3 = {inedge.src.t_op→r_op, inedge.comm}
11                 e4 = {r_op→vertex.t_op, 1to1}
12                 dag.add_edges([e1, e2, e3, e4])
13             else
14                 e = {inedge.src.t_op→t_op, 1to1}
15                 dag.add_edges([e])
16             if inedge.src.is_stateful() then
17                 m_op = MergeOp.create()
18                 dag.remove_edge(inedge)
19                 e1 = {inedge.src→m_op, inedge.comm}
20                 e2 = {m_op→vertex, 1to1}
21                 e3 =
                    {inedge.src.t_op→m_op,inedge.comm}
22                 e4 = {m_op→vertex, 1to1}
23                 dag.add_edges([e1, e2, e3, e4])
24     return dag
```

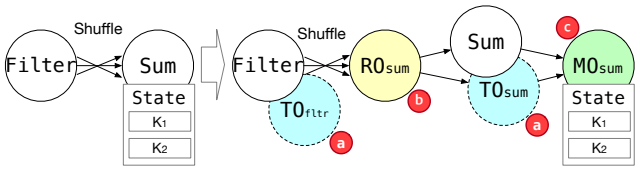


Figure 6: DAG transformation after graph rewriting.

4.2 Compile-time Graph Rewriting Algorithm

At the start of the application, our compiler applies the graph rewriting algorithm (Algorithm 1) to the application DAG, which produces a new DAG based on a set of conditional rules, as shown in Fig. 6. In our algorithm, TOs, ROs, and MOs are inserted as follows. *TOs* are cloned stream operators with additional features to run on SF instances, such as maintaining partial states for stateful operators. Since all original operators are potential candidates for offloading, we first create TOs for all operators (line 3, Fig. 6ⓐ). This way, all operators causing CPU bottlenecks can scale on SFs with TOs. *ROs* enable data communications between VM and SF instances when the communication pattern involves a shuffle or a broadcast (as one-to-one communications typically occur locally between pipelined operators) (§ 2). ROs run on existing VMs to redirect the input data to the downstream tasks running on either VMs or SFs without performing additional

computations (line 5-12, Fig. 6ⓑ). If the communication pattern involves the same number of partitions and tasks between two operators, we pipeline the corresponding TOs with a one-to-one edge (line 13-15). ROs incur almost no costs as they simply redirect events to the tasks on target instances (e.g., conventional or TO tasks). Lastly, we insert an *MO* after each stateful operator for every edge, so that the partially aggregated states on the TOs can be merged back into the original states on VMs (line 16-23, Fig. 6ⓒ), where the details of the merging mechanisms are provided in § 4.5. Stateless operators do not need to merge states, so they simply pass on their output to the following operators (e.g., filter operator in Fig. 6). During non-scaling periods, ROs are not activated and TOs and MOs do not receive any data, adding no computational costs to the runtime. The operators are only activated upon offloading actions.

4.3 Dynamic Offloading Policy

In this section, we describe when Sponge triggers offloading, how many SF instances it uses, and how many events it offloads. Our goal is to constantly maintain low query latencies while keeping CPU utilization stable across all active cloud instances. To achieve this, Sponge quickly calculates the total number of CPU cores needed to meet this goal and the Sponge scheduler redistributes the workload accordingly among the tasks placed on VMs and SFs.

Overall workflow. The Sponge runtime, shown in Fig. 5, is a main system component that performs monitoring of the resources and operator states to take immediate scaling actions as needed. Each executor continuously monitors CPU resources and input rates, typically every second, and observes if the CPU load falls outside a stable range for consecutive periods. If so, the Sponge runtime initiates the scaling phase by first calculating the target system throughput, based on the over-subscription period of CPUs and the current input rates (that jointly decide the number of events in the queue), and the recovery deadline (the time remaining to clear the events and return the system to a stable state). Subsequently, the Sponge runtime adds new SF instances as needed to meet the recovery deadline by sending requests to the Sponge scheduler. The number of new SF instances is chosen to be minimum to neither over-subscribe nor under-subscribe the active cloud instances, minimizing operational costs. After a scaling action is taken, the Sponge runtime returns to the monitoring phase. It is possible that the Sponge runtime may go through multiple monitoring-scaling phases before the system becomes stable.

4.3.1 Detailed Offloading Process

CPU utilization goals. Along with system metrics, such as the input rate and operator latency, Sponge measures the CPU load of the executor in order to maintain adequate CPU loads on individual nodes. Through extensive experiments, we have observed that the input rate r_i exceeds operator throughput m_i and event queues start to build up (i.e., $r_i > m_i$) when the CPU is occupied at around 75-80% of its capacity. We have also seen symptoms of over-provisioned system resources when the CPU load falls under 50-60%. Due to such reasons, we aim to maintain the CPU utilization range between 60-80%.

Events in the queue. Assuming $r_i(t) > m_i(t)$ between times t_p and t_{p+1} ($t_p < t_{p+1}$), the number of excess events accumulated in the queue can be formulated as $\int_{t_p}^{t_{p+1}} (r_i(t) - m_i(t))\,dt$. Obviously, the accumulated events in the queue will be smaller if the duration $d = t_{p+1} - t_p$ is smaller. This is the main reason for using SF instances over VMs – to reduce the duration of $r_i(t) > m_i(t)$.

Recovery deadline. Recovering from this event backpressure is achieved by providing the system with additional resources to achieve higher throughput, m_{i_o}. If additional resources are available from time t_o, we should set a deadline t_{o+1} until which we aim to empty the queue to return to a stable state for our streaming system. We base the deadline on the window interval of the query (e.g., 10 seconds) so that we can deliver the query results within the query's next output boundary. Then, the number of additional events that can be processed from the queue can be expressed as $\int_{t_o}^{t_{o+1}} (m_{i_o} - r_i(t))\,dt$, where the increased throughput is larger than the input rate ($m_{i_o} > r_i(t)$). As a result, we should adjust our throughput m_{i_o} with sufficient additional resources to meet our recovery deadline t_{o+1} (e.g., $t_{o+1} - t_o \leq 10$) such that the following equation holds:

$$\int_{t_p}^{t_{p+1}} (r_i(t) - m_i(t))\,dt \leq \int_{t_o}^{t_{o+1}} (m_{i_o} - r_i(t))\,dt \qquad (1)$$

The approximation of the integrals is based on Simpson's rule provided by [5], which turns complex calculations into simple arithmetic that incurs trivial overheads.

Stable throughput per VM CPU core. To calculate the target number of SF instances required to achieve our target throughput, we maintain records of the CPU usage rate of the VM node during stable loads. Assuming that a task p_i runs on a single VM core with an average usage rate of $\overline{u_{cpu_i^{VM}}}(\%)$ and an average task input rate of $\overline{r_i}[events/sec]$ based on the records, we scale and approximate the input rate and throughput for 100% utilization of the VM core $rpc_i^{VM}[events/(sec \cdot core)]$ for task p_i, as follows:

$$rpc_i^{VM} = \frac{\overline{r_i}}{\frac{\overline{u_{cpu_i^{VM}}}}{100}} \qquad [events/(sec \cdot core)] \qquad (2)$$

Required number of SF instances and data redistribution. Based on the approximation of how much input throughput a VM core can handle, we can calculate the number of required SF instances to achieve our target throughput m_{i_o} with a simple division. We offload tasks from VMs to SFs to keep the CPU utilization of VM clusters between $60-80\%$ in order to prevent resource over-provisioning. Hence, we target the VM CPU core usages at 70%, for our approximation to solidly fall

into our target with a ±10% buffer even when our profiling measurements exhibit minor errors on time-varying variables like $r_i(t)$.

Assuming the CPU capacity of each SF instance core is different from that of a VM core, we can derive a relation between them with profiling: $capa^{SF}_{core} = \rho * capa^{VM}_{core}$. Correspondingly, $rpc^{SF}_i = \rho * rpc^{VM}_i$ because the throughput is proportional to the CPU capacity. Altogether, the number of total SF cores c that we need to prepare to meet our latency goal for task p_i can be derived with Equation 2 as follows:

$$c = \lceil \frac{m_{i_o}}{0.7 \cdot rpc^{SF}_i} \rceil = \lceil \frac{m_{i_o} \cdot \overline{u_{cpu_s}}}{0.7 \cdot 100 \cdot \rho \cdot \bar{r}_i} \rceil \quad (3)$$

where the number of required SF instance cores increases as ρ decreases. Finally, the number of SF instances can be calculated with $\frac{c}{k}$ where k is the number of cores per SF instance ($k = 1$ in our evaluation).

When offloading stateless or stateful tasks, Sponge evenly redirects data or redistributes keys to the $\frac{c}{k}$ SF instances, while processing remaining events on VMs to keep 70% CPU usage in average. If the event distribution is skewed across the key space, the solution can be extended to use key histograms for more accurate key partitioning, as in existing approaches [13, 30]. Both during scaling up and down, the target CPU usage is set at 70% within our target range.

4.4 Reducing Cold Start Latency

In order to timely gain access to SFs, Sponge provides two options: (1) warm-up SF workers in advance by sporadically processing short events [21, 39, 58] to minimize the managed runtime initialization overheads [35], and (2) use solutions like AWS SnapStart [32], which bring shorter initialization times of SFs by taking a snapshot of the initialized SF instance environment and caching it for low-latency access [1]. As SFs are charged based on the memory usage time and the number of requests [9, 10, 22], prices for pre-warming SFs are trivial (nearly zero). By default, Sponge prepares and keeps enough SFs warm to handle up to 5× the stable load during the workload. For bursty loads that exceed 5× the stable load, Sponge timely prepares new instances with SnapStart [32] on AWS. SFs on AWS SnapStart [1, 32] show a slightly worse start-up time compared to the instances that are kept warm in advance, but the overhead is reduced by more than 80% compared to unoptimized JVM initializations, resulting in a sub-second total start-up time for SFs (§ 6.4). As a result, by enabling both methods for initializing the managed runtime, Sponge can timely gain access to SFs upon facing unpredictable bursty input loads.

4.5 Correctness

As stream systems are designed to be long-running, progress is tracked by the positions of the *watermarks* that flow along

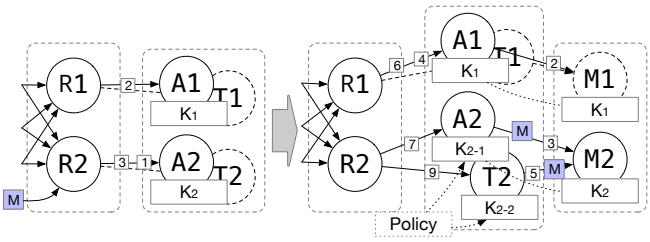


Figure 7: Once an operator (A2) tries to scale, an offload message (M) is generated at the RO (R2) to activate its TO (T2) and MO (M2). The offload message acts as a boundary among input events (1-9) for operator scaling and state merging.

with stream events [12, 15, 51, 57]. Based on the intuition, Sponge maintains correctness by (1) introducing a watermark in the event stream as a control message upon (de)activating an operator and (2) ensuring that all events between two watermarks are processed in the original system (i.e., without offloading) or on the offloaded operators [23, 36].

Concretely, upon detecting a possible bottleneck in an operator task p_i on an executor, Sponge fires a watermark message `M` into the data channel (Fig. 7). Upon receiving watermark messages, operators checkpoint their states to later recover from the checkpointed states, guaranteeing exactly-once processing. Sponge scales after temporarily pausing operators upon control messages and delivering messages to downstream tasks. Once all on-the-fly events in the data plane are consumed, downstream tasks send acks back to the upstream tasks to guarantee no event and state loss.

Thus, the events that arrive after `M` are immediately redirected to the tasks of the TOs on SFs, where partial states are aggregated if the operator is stateful. For stateful operators, TOs send the aggregated states to the following MOs placed on VMs, which know where to start merging the partial states with the original ones. Both incremental and appended aggregation can be mergeable with partial states, similar to how Flink [15] manages shared states, which causes moderate overhead on VMs. Even if events arrive out-of-order in the merge operators, they wait for the same watermark to arrive from the task in VM and its transient tasks so that the states can be synchronized. This ensures that all input data before and after `M` are processed according to the proposed optimizations.

5 Implementation

We have implemented Sponge with about 10K lines of Java with support for AWS Lambda, as follows:

Programming interface: To express a stream query as an application DAG, we use Apache Beam [12] application semantics, which is a widely used dataflow programming interface for various systems (e.g., Spark [7], Flink [15], Cloud-Dataflow [3]). In addition, as Beam provides APIs for developers to build associative and commutative operations (e.g.,

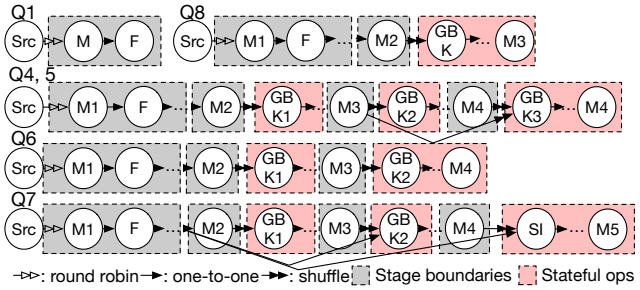


Figure 8: A simplified application DAG of stream queries used in our evaluation. `M` and `F` are map and filter operators, `GbK` is a stateful group-by-key operator for incremental aggregation on windows, and `SI` is a non-mergeable stateful operator for the join operation.

combiners), Sponge can extract this information to build the merge operators.

Compiler: Apache Nemo [51, 57] provides the intermediate representation and optimization pass abstractions, with which we can flexibly optimize application DAGs. We split our operator insertion into three separate optimization passes for inserting ROs, TOs, and MOs on Nemo to reshape the application DAG, defined by Apache Beam semantics.

Runtime: We modify the Nemo runtime [51, 57] to support the migration of tasks and the redirection of the data from VMs to SFs. Sponge executes worker processes on VM and SF instances, which each manages a thread pool that contains a fixed number of threads and assigns tasks to the threads. VM workers set up Netty [41] network channels and communicate with other VMs and SF workers, while there are no network channels set up between SF instances due to the communication constraint. For launching new VM and SF instances, as well as for deploying the worker code on AWS Lambda, we use boto3, the AWS SDK API for controlling AWS instances [14].

6 Evaluation

In our evaluation, we observe Sponge performance compared to other scaling mechanisms (§ 6.2), distinguish the factors that contribute to the Sponge performance (§ 6.3), compare the cold start latency reduction mechanisms (§ 6.4), and observe the latency-cost trade-off between SFs and VMs (§ 6.5).

6.1 Methodology

Environment. We use AWS EC2 r5.xlarge instances (32*GB* of memory and 4 vCores) as VM workers, and AWS Lambda instances as SF workers. As AWS Lambda offers one vCPU per 1,769*MB* and provides constant network bandwidth (i.e., $\sim$ 100Mbps) regardless of the instance size, we use single-core SF instances of 1,769*MB* to provision each instance with enough network bandwidth to achieve the throughput of the CPU core. VMs generally provide 10Gbps networks, which effortlessly cover the traffic generated by the CPU

Query	Stateful	State Size	# of Tasks (per Op.)	Stable input rate
Q1	X	-	120	550 K/s
Q4	O	~90 MB	60	190 K/s
Q5	O	~2.4 GB	70	19 K/s
Q6	O	~73 MB	70	230 K/s
Q7	O	~1.5 GB	90	15 K/s
Q8	O	~7 GB	60	60 K/s

Table 1: Characteristics of different NEXMark stream queries.

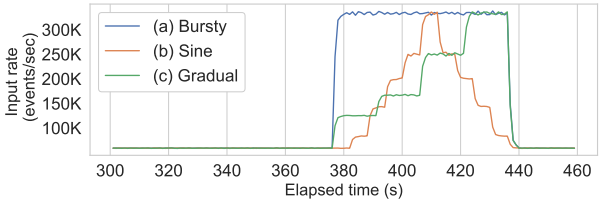


Figure 9: Examples of different bursty input patterns used in some experiments, where input rates increase at time $t = 380$. (a) shows a sudden increase from $60K$ to $300K$ ($5\times$) for 60 seconds, (b) shows a sine-curve increase and decrease, and (c) shows a gradual increase.

core throughput (i.e., $< 10\%$ bandwidth utilization when offloading 450K events/sec). We set up Amazon Virtual Private Cloud (VPC) for the data communication between the VM and SF instances for stable network connections.

Workloads. NEXMark [42] is a widely used streaming benchmark [28, 33] that analyzes auctions and bid data streams. NEXMark contains diverse stream queries with complex dataflow and stateful operations. Among the 8 (Q1-8) NEXMark queries, we choose 6 queries as shown in Table 1 because they represent distinctive data communication patterns and stateless and stateful operations. We omit Q2-3 because Q2 is a stateless query similar to Q1, and Q3 is a non-associative stateful query like Q7.

Fig. 8 illustrates the simplified original DAG of NEXMark queries, and Table 1 summaries the characteristics of the queries. The queries except for Q1 contain windowed operations. We configure the window size of queries as 60 seconds and the window interval as 1 second. While the system throughput declines with larger and more frequent windows, we evaluate under a frequent window interval to test Sponge under requirements for frequent, time-critical resource scaling. The throughput of the evaluated engine [51, 57] is similar to the performance of other stream processing engines [7, 15] when the same window size and interval are used. Nonetheless, in our evaluation, the bursty traffic is increased by up to $10\times$ events/sec and represents a wide range of realistic input rates in the field (§ 6.2).

Baseline. We compare Sponge with the following baselines:

- **NoScaling** executes stream queries on static VMs without scaling out stream queries.
- **VMBase** dynamically creates new VMs and migrates tasks

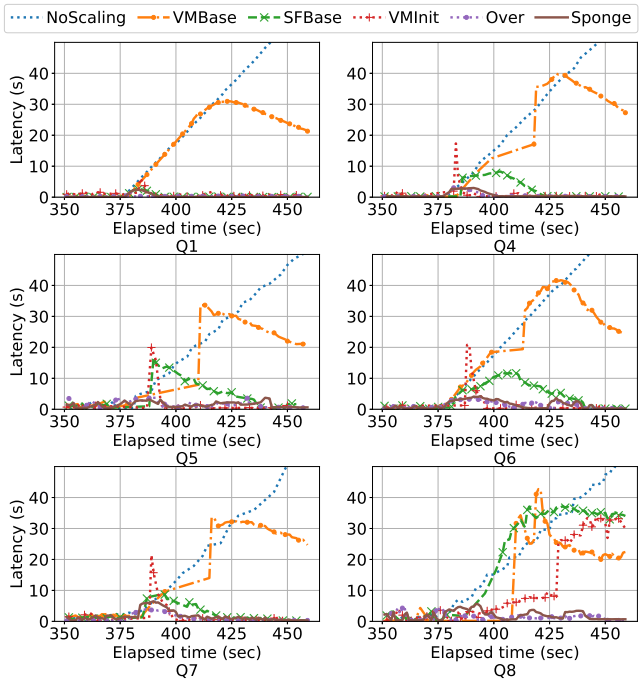


Figure 10: The tail latency graph, under a bursty load (Fig. 9(a)) at $t = 380s$ and scaling is triggered at $t = 381s$.

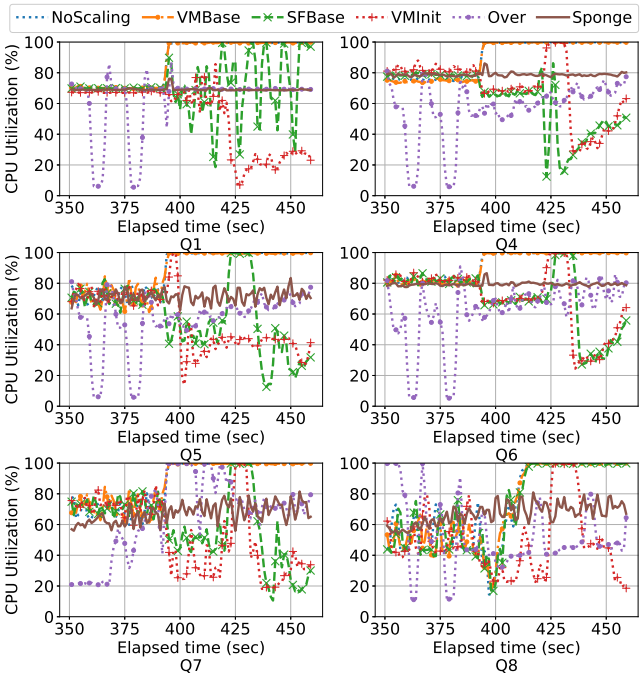


Figure 11: The CPU utilization graph, under a bursty load (Fig. 9(a)) at $t = 380s$ and scaling at $t = 381s$.

to the new VMs for scaling without dataflow reshaping.

• **SFBase** dynamically creates SF instances and migrates tasks to SFs for scaling without dataflow reshaping. For SFBase and Sponge, we prevent cold start latencies on SF workers as described in § 4.4.

• **VMInit** initializes new VMs in advance and migrates tasks to the new VMs for scaling without dataflow reshaping.

• **Over** over-provisions VMs and already has enough resources to cover input loads without dataflow reshaping.

Bursty traffic and resource allocation. We emulate bursty traffic by increasing the input rate over a short period of time, as shown in Fig. 9. In this traffic pattern, we first generate stable input streams where the input rate is stable and does not fluctuate. At a specific point (e.g., $t = 380s$ in our evaluation), we increase the input rate for a short period of time (e.g., 60 seconds) to emulate an increased load and then decrease the rate back to the stable input rate. In § 6.2, we observe the average performance of the different systems under up to $10\times$ burstiness ($\frac{\text{bursty input rate}}{\text{stable input rate}}$), and we provide detailed analysis on the effects of the burstiness rising from $3\times$ to $6\times$ in § 6.3. By default, the burstiness is set to $5\times$, as it distinctly shows the limitations of existing approaches comparatively. For example, as the stable CPU load is kept at $60-80\%$, most baselines already experience high latencies from $2\times$ burstiness, but the performance results are more clearly distinguishable between the baselines under the $5\times$ burstiness.

During the stable load, we run 5 VM workers. We generate events (per second) for the stable load such that all 5 VM workers undergo CPU usage between 60% and 80%, preventing the VM cluster from being under-loaded or overloaded. As queries have different computational complexity, the stable input rate is configured differently for each query as shown in the last column of Table 1. Once bursty loads occur, we dynamically allocate up to 200 single-core SF instances for *Sponge* and *SFBase*, and up to $50(10\times)$ new extra VM instances for *VMBase* depending on the query load.

6.2 Performance Analysis

Fig. 10 and Fig. 11 illustrate the 99^{th}-percentile tail latency and CPU utilization, respectively, of the different systems across different queries for the Burst traffic pattern in Fig. 9. Overall, *Sponge* and *Over* exhibit lower latencies compared to others during bursty periods and successfully keep the CPU utilization stable. The latency of *NoScaling* continuously increases with full CPU utilization as the existing VMs are overloaded and never perform offloading. Henceforth, we discuss Sponge and other baselines that perform scaling. For SF-based strategies that are restricted by the prohibited direct communication between SF instances, we profile their operator costs and manually configure them to make the best scaling decisions.

Sponge. Sponge reduces the tail latency on average by 88% compared to *VMBase* and 70% compared to *SFBase* and performs comparably to *Over*. Sponge also keeps the CPU utilization relatively stable across time, as shown in Fig. 11. Subsequently, we illustrate below why other scaling strategies cannot deliver the same benefits as Sponge in further detail.

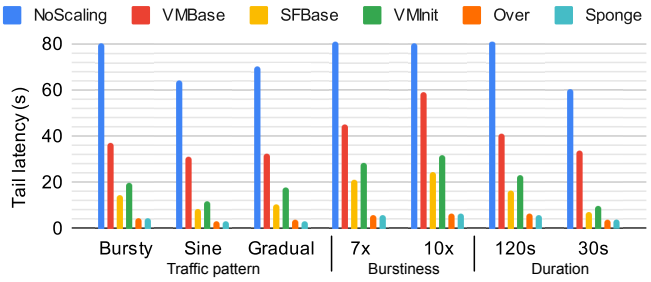


Figure 12: Summarized results of the experiments, with similar settings as in Fig. 10, displaying the average peak tail latency across the different NEXMark queries under diverse input patterns and burstiness.

VMBase. The latency of *VMBase* in Fig. 10 increases by at least 30s due to the slow start-up time of the VMs. Specifically, we observe that it takes around 25-30 seconds for the VMs to start, and around 4 extra seconds for managed runtime (i.e., JVM) worker processes to start on the newly started VMs. Moreover, as JVM processes are cold at the beginning and JIT compilation is not triggered, the processing throughput is low in the beginning, which causes extra latency of up to 44 seconds. After new VMs are instantiated, tasks are migrated to new VMs, and the latency of the VM decreases as the throughput eventually becomes larger than the input rate. Nevertheless, the CPU utilization of *VMBase* shown in Fig. 11 is continually kept high after the peak load, as it tries to climb down from the latency peak by heavily processing the data in the event queue.

SFBase. The slow start-up time of VMs can be mitigated by using SFs as shown in *SFBase*. Upon scaling out Q1 (a simple stateless query), *SFBase* significantly reduces the latency and CPU compared to *VMBase*, as the start-up time of an SF instance only takes a few hundred milliseconds in our evaluation. This result suggests that only by using SFs instead of VMs, we can significantly improve the latency for scaling out a simple stateless query, similar to MArk which handles bursty loads of stateless inference jobs [58].

However, for scaling out other complex queries with N-to-N shuffle data communications and stateful operations, the performance gain of *SFBase* compared to *VMBase* declines. It indicates that naïvely scaling queries on SFs without any operator insertion has limitations due to the challenges explained in § 3. In Q4 and Q6, latency increases by up to 12 seconds because the operators with shuffle edges cannot be redistributed to SFs and VMs become the bottleneck. In Q5, Q7, and Q8, latency and CPU spikes are caused by task and state migration overheads.

VMInit. Like *SFBase*, *VMInit* reduces the slow start-up time of VMs by starting them in advance. For *VMInit*, queries with N-to-N shuffle data communications can be offloaded, but we can see that it still incurs task and state migration overheads resulting in short steep peaks of tail latencies and CPU usage, which is highlighted in Q8.

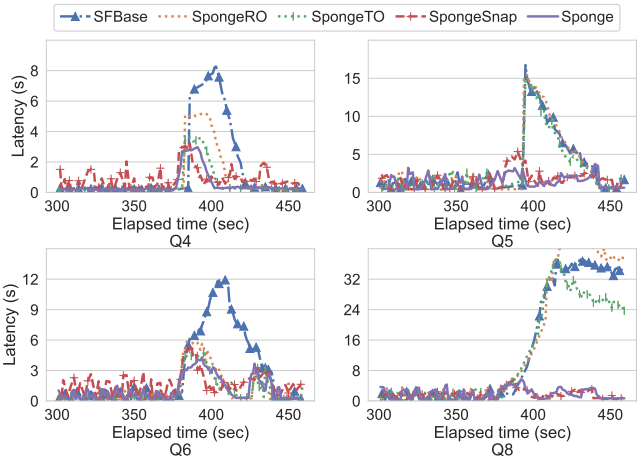


Figure 13: The latency graphs for SF, *SpongeRO*, *SpongeTO*, SpongeSnap, and Sponge to analyze and break down the performance improvements of Sponge.

Over. The over-provisioned case is the most expensive solution, providing enough resources for the peak loads without considering an upper bound for runtime costs. In Fig. 10, we can see a slight increase in latency as the input load increases, but it soon stabilizes back. The CPU usage in Fig. 11 displays an under-utilization before the peak load, but shows an adequate utilization rate afterward, as it is allocated with an adequate amount of resources for the peak load.

Input patterns. In Fig. 12, we can see the average tail latency among the different queries along the different input patterns. We can see that *Sponge* and *Over* show good performance among all settings, and *NoScaling* continuous increases in most cases. The sine and gradual bursts show a relatively mild effect compared to others, as their bursts are more gentle. We can see that while 120s and 30s bursts show somewhat similar results, 7× and 10× bursts show higher tail latencies due to the increased load.

6.3 Graph Rewriting Effect

To validate our design, we analyze the performance gain on Sponge with the following additional baselines:

- **SpongeRO** scales queries on SFs while allowing direct communication between SF instances with ROs only.
- **SpongeTO** scales queries on SFs by adding event redirection atop *SpongeRO* with ROs *and* TOs.
- **SpongeSnap** shows performance for Sponge, with ROs, TOs, and MOs, on SnapStart, without pre-warming instances.

Fig. 13 illustrates the tail latencies of *SFBase*, *SpongeRO*, *SpongeTO*, *SpongeSnap*, and *Sponge* in more detail. Q1 and Q7 are omitted in the figure, as Q1 is a simple stateless query, and Q7 is represented by Q5 and Q8.

Router operator effect. Comparing *SpongeRO* with *SFBase* shows the effect of router operators. In Q4 and Q6, SFs exhibit higher latencies as VMs are bottlenecked while processing

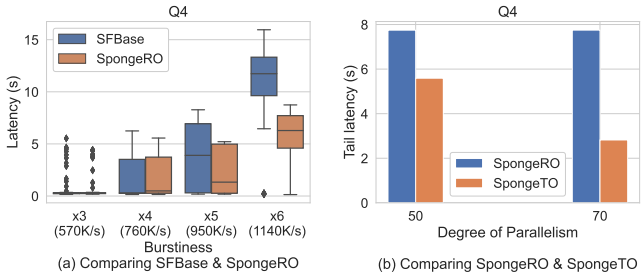


Figure 14: Comparison on Q4 for (a) *SFBase* and *SpongeRO* on diverse burstiness, and (b) *SpongeRO* and *SpongeTO* on different degrees of parallelism (# of parallel tasks).

events for `M` operators (Fig. 8) on VMs (only 3% of input events are filtered before `M2`). As naive SFs can only offload one of `M` and `GbK`, we choose to offload `GbK`, as the amount of computation on `M` is smaller than that of `GbK` due to the additional aggregation. However, the input rate of `M` on VMs becomes higher than the maximum throughput on the VMs with 5× bursts in Q4 and Q6, and events pile up in `M` operators, incurring latency increases in SFs. In Q5 and Q8, the latency of *SFBase* is similar to *SpongeRO* as VMs sufficiently handle the load on `M` operators. The main bottlenecks in Q5 and Q8 are `GbK` operators, which incur large aggregate computations. This result indicates that the RO is effective when the input rate and the overhead caused by the operators running on VMs are high.

We also evaluate when VMs become bottlenecks on `M` operators, by varying the burstiness ($\frac{\text{bursty input rate}}{\text{stable input rate}}$ from 3 to 6 in Q4 (Fig. 14(a)). In the figure, the bottom and top of the box are the 25^{th} and 75^{th} percentiles, the line indicates the median, error bars are the 95% confidence interval, and outliers are dotted as rhombi. VMs sufficiently handle 3× and 4× burstiness, and the latency of *SFBase* does not increase and is similar to *SpongeRO*. However, when the burstiness increases to 6×, VMs become the bottleneck in processing the input events of `M`. Unlike *SFBase*, *SpongeRO* adds an RO between `M` and `GbK`, and migrates both `M` and `GbK` to SFs while keeping the RO on VMs. As an RO does not (de)serialize events and does not perform computation, the amount of computation of RO is always smaller than that of `M`, and reduces latencies by up to 70%.

Transient operator effect. Transient operators enable Sponge to redirect data without stopping the workload for rescheduling. The effectiveness of transient operators increases as the number of tasks to be migrated (or redirected) increases. Q4 requires a large number of tasks to be migrated or redirected. For Q4, we had to migrate or redirect 85% of total tasks to SFs to mitigate the bottleneck in the VMs in *SpongeRO* and *SpongeTO*. *SpongeRO* takes around 2.8 seconds for migrating its tasks. In contrast, *SpongeTO* takes around 1.4 seconds for redirecting its tasks. Due to the fast redirection mechanism, *SpongeTO* additionally reduces the latency by up to 28% compared to *SpongeRO*.

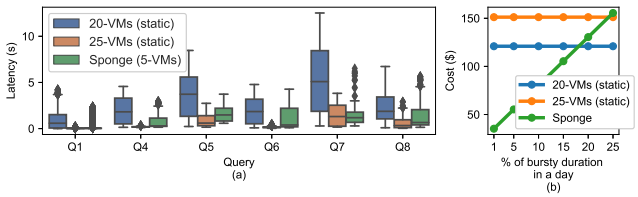


Figure 15: (a) The latency during a bursty period, and (b) a rough calculation of the cost according to the % of the bursty duration throughout the day.

When the degree of parallelism increases, the number of tasks to be migrated or redirected also increases. Fig. 14(b) illustrates the tail latency under different degrees of parallelism in Q4. With 50 parallel tasks for each operator, the task migration/redirection overhead is small, but the latency increases after the migration and redirection in both *SpongeRO* and *SpongeTO*, as a smaller degree of parallelism makes the system more prone to unevenly skewed tasks. With 70 parallel tasks, the overall latency decreases but the task migration overhead increases in *SpongeRO*. As a result, the peak latency increases by up to 8 seconds. In contrast, due to the lightweight redirection optimization, the peak latency of *SpongeTO* is kept at around 3.5 seconds, which is 56% smaller than *SpongeRO*.

Merge operator effect. Even with ROs and TOs, *SpongeTO* still suffers from high latencies in Q5 and Q8 due to the state encoding/decoding overheads. The state migration overhead is trivial in Q4 and Q6 (< 100MB), but the overhead increases with the state size. The time to encode/decode the states of Q5 and Q8 takes around 13s (for ∼2.4GB state) and 35s (for ∼ 7 GB state), respectively. As a result, the latency of *SpongeTO* increases by up to 15 and 38 seconds in Q5 and Q8. In contrast, Sponge significantly reduces the latencies to 4 seconds in Q5, and to 6 seconds in Q8, preventing state migration overheads with MOs.

6.4 Cold Start Latency Reduction Methods

In § 4.4, we describe two methods for reducing the cold start latency: by keeping warm SF instances and by using snapshots of SFs through tools like SnapStart [32]. In Fig. 13, we can see that the performance of *SpongeSnap*, which solely bases its initialization method on SnapStart [32], is slightly worse, but comparable with *Sponge*, which uses a hybrid of both methods in optimizing the managed runtime (e.g., JVM) initialization overhead. Since the overhead is reduced by more than 80% with SnapStart [32], and > 90% with warm SF instances compared to the original managed runtime initialization methods on SF instances, both methods succeed to timely supply SFs within a sub-second total start-up time.

6.5 Latency-Cost Trade-Off

The cost of using SF instances may be higher than overprovisioning VMs when the bursty input persists. In such cases, it makes more sense to launch new VMs while Sponge

handles the bursty traffic and offload our tasks to the VM. To investigate the latency-cost trade-off and figure out when it is more beneficial to launch new VMs, we compare the following two VM over-provisioning approaches with Sponge in terms of latency and cost on the workload shown in Fig. 9(a). One is *20-VMs (static)*, where 20 VMs are statically allocated without dynamic scaling, and the other is *25-VMs (static)*, where 25 VMs are statically allocated. As the default number of VMs used in *Sponge* is 5, *20-VMs* and *25-VMs* allocate 4× and 5× more VMs compared to *Sponge*, respectively.

Fig. 15(a) illustrates the latency of *20-VMs*, *25-VMs*, and *Sponge* during the bursty period. The latency of *Sponge* is in between *20-VMs* and *25-VMs*. Compared to *25-VMs*, which has enough resources to handle 5× bursty loads, Sponge has inherent scaling overheads due to the redirection and migration protocols. This is why the latency of *Sponge* is slightly higher than *25-VMs*.

In terms of cost, Fig. 15(b) shows a rough calculation of cost according to the bursty duration in a day. For instance, 1% of bursty duration represents that bursty loads happen for $24hr * 0.01$ during a day. Basically, the cost of *Sponge* is smaller than others when bursty loads occur in short durations. When the duration of the bursty load is less than 15%, *Sponge* has lower latency and cost compared to *20-VMs*. When the bursty load persists (at more than 25% in Fig. 15(b)), the cost of *Sponge* exceeds *25-VMs* due to the high cost of the SF instances. In this case, it is more beneficial to statically over-provision VM resources in terms of latency and cost. Nevertheless, as presented in existing works [36, 40], bursty loads are mostly short-lived, and persistent peaks are comparatively much rare, resulting in their duration generally falling much below 15% of the total time. Sponge provides mechanisms to initially provide prompt scaling with fast-starting SFs regardless of the peak duration and later expands the cluster to additional slow-starting VMs if the peaks persist, making the solution effective with any bursty traffic in terms of both cost and latency.

7 Related Work

To the best of our knowledge, Sponge is the first work that addresses all technical challenges described in § 3. There are some existing works that partially address the challenges, and we compare them with Sponge.

Data communication across SF instances. Researchers have exploited fast-starting SF instances for various workloads such as interactive data analytics [27, 44], video analytics [4, 21], and daily applications [20]. These applications are also represented as DAGs, and shuffle operations are required between SF instances. Their solutions to enable data communication across SF instances enable using additional VM relay servers [21], using HDFS in VMs [27], building an ephemeral storage service [31], and using a NAT-traversal technique [20]. Sponge router operators enable data communication across SF instances preserving event-based stream processing with low latency, without requiring any of the additional VM resources or NAT-traversal technique.

Optimizing state migration. Rhino [18] and ChronoStream [55] replicate states of stream queries across extra (over-provisioned) machines to minimize state migration overheads. Replicating and holding states requires costly over-provisioning of long-running resources like VMs. Holding states on SFs will cause additional state recovery and cost when SFs are reclaimed by cloud vendors. Megaphone [25] proposes fluid migration that smoothly migrates states from source to destination resources for a long period to reconfigure system configurations. However, when bursty loads happen, the reconfiguration must be executed in a short period of time. As a result, a large amount of state migration is inevitable to quickly migrate the load on VMs. In contrast, Sponge avoids state migration from VMs to SFs by just forwarding data to SFs and merging partial states in SFs into the existing VMs.

Scaling policy. Regarding scaling policies, SEEP [16], StreamCloud [24], and Dhalion [19] use metrics like CPU utilization for their decisions. Systems such as DS2 [28] aim to measure the processing and output rates of individual dataflow operators through system instrumentation. Many of these scaling policies are designed to be agnostic to the underlying scaling mechanisms and resource acquisition schemes. In contrast, the Sponge scaling policy also explicitly considers the characteristics of SF instances and offloads the right amount of computations to keep the CPU utilization high.

8 Conclusion

Sponge harnesses SF instances for offloading bursty loads from existing VMs in streaming workloads. Sponge minimizes task migration overheads and addresses data communication constraints on SF instances by inserting new stream operators in the application DAG: router, transient, and merge operators. Sponge also provides an offloading policy that determines when and how to offload the increased input loads. Our evaluation on AWS EC2 and Lambda shows that the Sponge operators are effective in significantly reducing tail latencies in stream processing upon unpredictable bursty loads, compared to existing scaling mechanisms on VMs and SFs.

Acknowledgments

We thank our shepherd Maria Carpen-Amarie and the anonymous reviewers for their feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics), the 2023 Research Fund (1.230019) of UNIST, and Electronics and Telecommunications Research Institute(ETRI) grant funded by the Korean government [23ZS1300].

References

[1] AGACHE, A., BROOKER, M., FLORESCU, A., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (USA, 2020), NSDI'20, USENIX Association, p. 419–434.

[2] AKIDAU, T., BALIKOV, A., BEKIROĞLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. MillWheel: fault-tolerant stream processing at internet scale. *VLDB Journal 6*, 11 (2013), 1033–1044.

[3] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNÁNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., AND WHITTLE, S. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *VLDB Journal 8*, 12 (2015), 1792–1803.

[4] AO, L., IZHIKEVICH, L., VOELKER, G. M., AND PORTER, G. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, Association for Computing Machinery, p. 263–274.

[5] Apache Commons Math. https://commons.apache.org/proper/commons-math/, 2023.

[6] ARASU, A., CHERNIACK, M., GALVEZ, E., MAIER, D., MASKEY, A. S., RYVKINA, E., STONEBRAKER, M., AND TIBBETTS, R. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30* (2004), pp. 480–491.

[7] ARMBRUST, M., DAS, T., TORRES, J., YAVUZ, B., ZHU, S., XIN, R., GHODSI, A., STOICA, I., AND ZAHARIA, M. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, Association for Computing Machinery, p. 601–613.

[8] AWS. Configuring Lambda function options, 2023. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html.

[9] AWS Lambda. https://aws.amazon.com/lambda, 2023.

[10] Azure Function. https://docs.microsoft.com/en-us/azure/azure-functions/, 2023.

[11] Various Traffics in the Cloud. https://intercept.cloud/en/news/checklist-part-1-choose-your-strategy-before-you-migrate-to-azure/, 2023.

[12] Apache beam. https://beam.apache.org/.

[13] BINDSCHAEDLER, L., MALICEVIC, J., SCHIPER, N., GOEL, A., AND ZWAENEPOEL, W. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, Association for Computing Machinery.

[14] AWS SDK for Python (Boto3). https://boto3.amazonaws.com/v1/documentation/api/latest/index.html, 2023.

[15] CARBONE, P., KATSIFODIMOS, A., EWEN, S., MARKL, V., HARIDI, S., AND TZOUMAS, K. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull. 38*, 4 (2015), 28–38.

[16] CASTRO FERNANDEZ, R., MIGLIAVACCA, M., KALYVIANAKI, E., AND PIETZUCH, P. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, Association for Computing Machinery, p. 725–736.

[17] Stream Processing with IoT Data: Challenges, Best Practices, and Techniques. https://www.confluent.io/blog/stream-processing-iot-data-best-practices-and-techniques/, 2023.

[18] DEL MONTE, B., ZEUCH, S., RABL, T., AND MARKL, V. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 2471–2486.

[19] FLORATOU, A., AGRAWAL, A., GRAHAM, B., RAO, S., AND RAMASAMY, K. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment 10*, 12 (2017), 1825–1836.

[20] FOULADI, S., ROMERO, F., ITER, D., LI, Q., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 475–488.

[21] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 363–376.

[22] Google Cloud Function. https://cloud.google.com/functions/docs/, 2023.

[23] GU, R., YIN, H., ZHONG, W., YUAN, C., AND HUANG, Y. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 539–556.

[24] GULISANO, V., JIMENEZ-PERIS, R., PATINO-MARTINEZ, M., SORIENTE, C., AND VALDURIEZ, P. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems 23*, 12 (2012).

[25] HOFFMANN, M., LATTUADA, A., AND MCSHERRY, F. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment 12*, 9 (2019), 1002–1015.

[26] ISLAM, S., VENUGOPAL, S., AND LIU, A. Evaluating the Impact of Fine-Scale Burstiness on Cloud Elasticity. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, Association for Computing Machinery, p. 250–261.

[27] JAIN, A., BAARZI, A. F., KESIDIS, G., URGAONKAR, B., ALFARES, N., AND KANDEMIR, M. SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS. In *Proceedings of the 21st International Middleware Conference* (New York, NY, USA, 2020), Middleware '20, Association for Computing Machinery, p. 236–250.

[28] KALAVRI, V., LIAGOURIS, J., HOFFMANN, M., DIMITROVA, D., FORSHAW, M., AND ROSCOE, T. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 783–798.

[29] KALIM, F., XU, L., BATHEY, S., MEHERWAL, R., AND GUPTA, I. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), pp. 249–262.

[30] KE, Q., ISARD, M., AND YU, Y. Optimus: A Dynamic Rewriting Framework for Data-Parallel Execution Plans. In *Eurosys 2013* (April 2013), ACM.

[31] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[32] Lambda SnapStart. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html, 2023.

[33] LI, S., GERVER, P., MACMILLAN, J., DEBRUNNER, D., MARSHALL, W., AND WU, K.-L. Challenges and Experiences in Building an Efficient Apache Beam Runner for IBM Streams. *Proc. VLDB Endow. 11*, 12 (aug 2018), 1742–1754.

[34] LIN, M., WIERMAN, A., ANDREW, L. L. H., AND THERESKA, E. Dynamic right-sizing for power-proportional data centers. In *2011 Proceedings IEEE INFOCOM* (2011), pp. 1098–1106.

[35] LION, D., CHIU, A., SUN, H., ZHUANG, X., GRCEVSKI, N., AND YUAN, D. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (USA, 2016), OSDI'16, USENIX Association, p. 383–400.

[36] MAI, L., ZENG, K., POTHARAJU, R., XU, L., SUH, S., VENKATARAMAN, S., COSTA, P., KIM, T., MUTHUKRISHNAN, S., KUPPA, V., DHULIPALLA, S., AND RAO, S. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proceedings of the VLDB Endowment* (2018), 1303–1316.

[37] MI, N., CASALE, G., CHERKASOVA, L., AND SMIRNI, E. Injecting Realistic Burstiness to a Traditional Client-Server Benchmark. In *Proceedings of the 6th International Conference on Autonomic Computing* (New York, NY, USA, 2009), ICAC '09, Association for Computing Machinery, p. 149–158.

[38] MIAO, H., JEON, M., PEKHIMENKO, G., MCKINLEY, K. S., AND LIN, F. X. StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery, p. 167–181.

[39] MÜLLER, I., MARROQUÍN, R., AND ALONSO, G. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 115–130.

[40] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., AND ROWSTRON, A. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)* (San Diego, CA, Dec. 2008), USENIX Association.

[41] Netty. http://netty.io/, 2023.

[42] Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/, 2023.

[43] POTHARAJU, R., KIM, T., WU, W., ACHARYA, V., SUH, S., FOGARTY, A., DAVE, A., RAMANUJAM, S., TALIUS, T., NOVIK, L., AND RAMAKRISHNAN, R. Helios: Hyperscale Indexing for the Cloud & Edge. *Proc. VLDB Endow. 13*, 12 (Aug 2020), 3231–3244.

[44] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 193–206.

[45] RAJADURAI, S., BOSBOOM, J., WONG, W.-F., AND AMARASINGHE, S. Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, Association for Computing Machinery, p. 98–112.

[46] RASTEGAR, S. H., ABBASFAR, A., AND SHAH-MANSOURI, V. Rule Caching in SDN-Enabled Base Stations Supporting Massive IoT Devices With Bursty Traffic. *IEEE Internet of Things Journal 7*, 9 (2020), 8917–8931.

[47] ROBINSON, B., POWER, R., AND CAMERON, M. A Sensitive Twitter Earthquake Detector. In *Proceedings of the 22nd International Conference on World Wide Web* (New York, NY, USA, 2013), WWW '13 Companion, Association for Computing Machinery, p. 999–1002.

[48] SANDUR, A., PARK, C., VOLOS, S., AGHA, G., AND JEON, M. Jarvis: Large-scale Server Monitoring with Adaptive Near-data Processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)* (2022), IEEE, pp. 1408–1422.

[49] SHAH, M., HELLERSTEIN, J., CHANDRASEKARAN, S., AND FRANKLIN, M. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)* (2003), pp. 25–36.

[50] SONG, W. W., JEON, M., AND CHUN, B.-G. SWAN: WAN-Aware Stream Processing on Geographically-Distributed Clusters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2022), APSys '22, Association for Computing Machinery, p. 78–84.

[51] SONG, W. W., YANG, Y., EO, J., SEO, J., KIM, J. Y., LEE, S., LEE, G., UM, T., CHO, H., AND CHUN, B.-G. Apache Nemo: A Framework for Optimizing Distributed Data Processing. *ACM Transactions on Computer Systems (TOCS) 38*, 3-4 (2021), 1–31.

[52] UM, T., LEE, G., AND CHUN, B.-G. Pluto: High-performance iot-aware stream processing. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)* (2021), pp. 79–91.

[53] VENKATARAMAN, S., PANDA, A., OUSTERHOUT, K., ARMBRUST, M., GHODSI, A., FRANKLIN, M. J., RECHT, B., AND STOICA, I. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 374–389.

[54] WANG, L., FU, T. Z. J., MA, R. T. B., WINSLETT, M., AND ZHANG, Z. Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing. In *the ACM International Conference on Management of Data conference (SIGMOD)* (2019), ACM.

[55] WU, Y., AND TAN, K.-L. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering* (2015), pp. 723–734.

[56] XU, D., LIU, X., AND VASILAKOS, A. V. Traffic-aware resource provisioning for distributed clouds. *IEEE Cloud Computing 2*, 1 (2015), 30–39.

[57] YANG, Y., EO, J., KIM, G.-W., KIM, J. Y., LEE, S., SEO, J., SONG, W. W., AND CHUN, B.-G. Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 177–190.

[58] ZHANG, C., YU, M., WANG, W., AND YAN, F. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 1049–1062.

On-demand Container Loading in AWS Lambda

Marc Brooker
Amazon Web Services

Mike Danilov
Amazon Web Services

Chris Greenwood
Amazon Web Services

Phil Piwonka
Amazon Web Services

Abstract

AWS Lambda is a serverless event-driven compute service, part of a category of cloud compute offerings sometimes called Function-as-a-service (FaaS). When we first released AWS Lambda, functions were limited to 250MB of code and dependencies, packaged as a simple compressed archive. In 2020, we released support for deploying container images as large as 10GiB as Lambda functions, allowing customers to bring much larger code bases and sets of dependencies to Lambda. Supporting larger packages, while still meeting Lambda's goals of rapid scale (adding up to 15,000 new containers per second for a single customer, and much more in aggregate), high request rate (millions of requests per second), high scale (millions of unique workloads), and low start-up times (as low as 50ms) presented a significant challenge.

We describe the storage and caching system we built, optimized for delivering container images on-demand, and our experiences designing, building, and operating it at scale. We focus on challenges around security, efficiency, latency, and cost, and how we addressed these challenges in a system that combines caching, deduplication, convergent encryption, erasure coding, and block-level demand loading.

Since building this system, it has reliably processed hundreds of trillions of Lambda invocations for over a million AWS customers, and has shown excellent resilience to load and infrastructure failures.

1 Introduction

AWS Lambda is a serverless event-driven compute service, part of a category of cloud compute offerings sometimes called Function-as-a-service (FaaS). First launched in 2015, today AWS Lambda functions run millions of times per second over millions of unique customer workloads. One factor that attracts customers to Lambda is its ability to scale up to handle increased load, typically in less than one second (and often as quickly as 50ms). This scale-up time, which customers have come to refer to as *cold-start time*, is one of the most important metrics that determine the customer experience in FaaS systems. When we launched AWS Lambda, we recognized that reducing data movement during these cold starts was critical. Customers deployed functions to Lambda in compressed archives (*.zip* files), which were unpacked as each function instance was provisioned. As Lambda evolved, and customers increasingly looked to deploy more complex applications, there was significant demand for larger deployments, and the ability to use container tooling (such as *Docker*) to create and manage these deployment images. Customers also wanted Lambda to support these images without compromising on cold-start performance.

Adding container support to AWS Lambda without regressing on cold-start time presented a significant technical challenge for our team. The core challenge is simply one of data movement. Today, Lambda can start up to 15,000 containers a second [18] for production workloads, and we expect to scale further for future workloads. Simply moving and unpacking a 10GiB image for each of these 15,000 containers would require 150Pb/s of network bandwidth. To achieve scalability and cold-start latency goals, we needed to take advantage of three factors which simplify this problem:

Cacheability While Lambda serves hundreds of thousands of unique workloads, large scale-up spikes tend to be driven by a smaller number of images, suggesting that the workload is highly cacheable.

Commonality Many popular images are based on common base layers (such as our own AWS base layers, or open source offerings like Alpine). Caching and deduplicating these common base layers reduce data movement for all containers that build on them.

Sparsity Most container images contain a lot of files, and file contents, that applications don't need at startup (or potentially never need). Harter et al [15] found that on average only 6.4% of container data is needed at startup.

Our solution combines caching, deduplication, erasure coding, and sparse loading to take advantage of our needs. With-

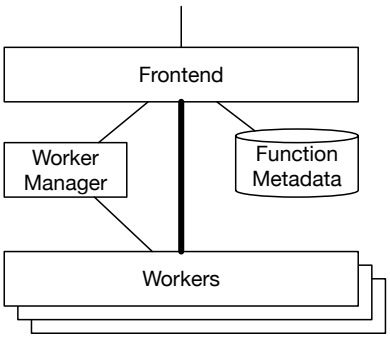


Figure 1: Architecture of the AWS Lambda invoke path

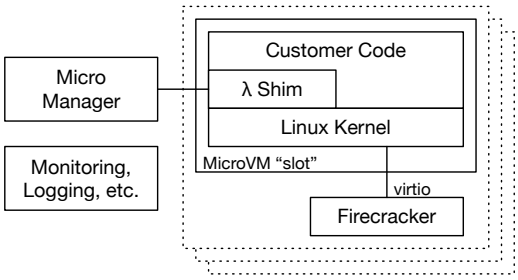


Figure 2: Architecture of the AWS Lambda worker

out adding any customer visible complexity (they simply upload a container image to a convenient repository), we were able to achieve our scale and cold-start latency goals, while having significant headroom for future scaling.

In this section, we present the existing architecture of AWS Lambda, and the overall architecture of our system. Section 2 presents the low-level implementation of our sparse loading solution. The cache architecture, and use of erasure coding to improve scalability and tail latency is presented in Section 4. Section 3 presents our convergent encryption-based secure deduplication architecture. Finally Section 6 compares our solution to other approaches from academia and industry.

1.1 Existing Architecture Overview

To reduce risk and optimize time-to-market, we wanted to introduce these new capabilities to Lambda with the minimum amount of change to the existing architecture, as shown in Figure 1. Requests to execute a certain function (we call these *invokes*) arrive via a load-balanced stateless frontend service. This service loads the metadata associated with the request, performs authentication and authorization, and then sends a request to the *Worker Manager*, requesting capacity. *Worker Manager* is a stateful, sticky, load balancer. For every unique function in the system, it keeps track of what capacity is available to run that function, where that capacity is in the fleet, and predicts when new capacity may be needed. If capacity is available, the Worker Manager instructs the frontend to forward the request payload to a Worker, where the function is executed. If no capacity is available, the Worker Manager identifies a Worker with available CPU and RAM, and sends a request to start a sandbox for the relevant function. Once this is complete, the frontend is notified and the function is executed.

Each Lambda worker, as shown in Figure 2, includes a small controller process, the Micro Manager, some additional agents for logging and monitoring, and a large number of MicroVMs. Each MicroVM, based on our Firecracker [3] hypervisor, contains the code for a single Lambda function for a single customer. Inside the MicroVM is a minimized Linux guest kernel, a small shim that provides Lambda's programming model, any provided runtime (e.g. the JVM for Java or CoreCLR for .NET), and the customer's code and libraries. As described in our Firecracker paper [3], the key concern here is security: customer code and data is not trusted, and the only communication between the workload inside the MicroVM and the shared worker components is over a simple, well tested, and formally verified implementation of *virtio* [27, 32] (specifically *virtio-net* and *virtio-blk*).

In the first generation architecture (before this work), when a new MicroVM is created with new capacity for a particular function, the Worker downloads the function image (a *.zip* file up to 250MiB in size) from Amazon S3, and unpacks it into the MicroVM guest's filesystem. This model is simple, and works well for small images, but requires the full archive to be downloaded and unpacked before the new MicroVM can do any work. To support larger images, we wanted to avoid this blocking download, and avoid the storage cost of unpacking the entire archive if only part of it is used.

2 Block-Level Loading

To take advantage of the *sparsity* property of containers, we needed to allow the system to load (and store) only the data the application needs, ideally at the time it needs it. Approaches like Slacker [15] and Starlight [8] have approached this problem at the filesystem level - a natural fit for containers, which are built as an overlayed stack of file-level archives. This approach isn't the right one for our environment. We believed that the inherent complexity of filesystems, and additional complexity of overlaying multiple filesystems, would unacceptably increase the attack surface of the shared components in Lambda. Instead, we decided to keep the block-level *virtio-blk* interface between the MicroVM guest and the hypervisor, perform all filesystem operations inside the guest. This requires performing sparse loading at the block, rather than file, level.

Figure 3 shows our high-level architecture, showing the Lambda worker (shown in detail in Figure 4) where customer's code is run, container registry which contains the primary copy of customer's container images, and the chunk

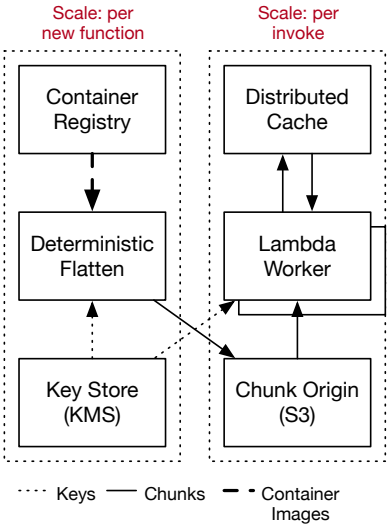


Figure 3: High-level system architecture.

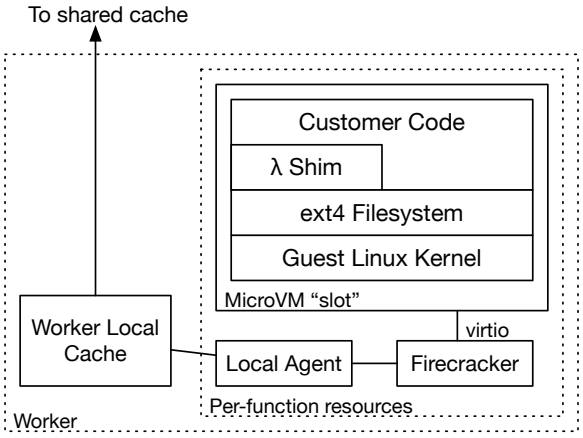


Figure 4: Lambda worker with per-worker, per-customer, and in-guest components

creation and caching infrastructure.

Our first step in supporting block-level loading is to collapse the container image into a block device image. As described in the OCI image specification [1], a container image is a stack of tarball *layers*. In the typical container stack, these layers are overlayed at at runtime using *overlayfs*. In our implementation, we perform this overlaying operation at the time the function is initially created, following a deterministic flattening process which applies each tarball in order to create a single *ext4* filesystem. Function creation is a low-rate control-plane process, that is typically only triggered by customers when they make changes to their code, configuration, or architecture. Even the most aggressive adoptees of continuous integration only make these changes on the order of minutes, while function invocation can happen up to millions of times a second.

The flattening process is designed so that blocks of the filesystem that contain unchanged files will be identical, allowing for block-level deduplication of the flattened images between containers that share common base layers. We'll revisit this in Section 3, but the high-level reason is that differences between functions (and even more so between versions of the same function) are typically much smaller than the functions themselves. The flattening process proceeds by unpacking each layer onto an *ext4* filesystem, using a modified filesystem implementation that performs all operations deterministically. Most filesystem implementations take advantage of concurrency to improve performance, introducing nondeterminism. Ours is serial, and deterministically chooses normally-variable parameters like modification times.

Following the flattening process, the flattened filesystem is broken up into fixed-size chunks, and those chunks are uploaded to the origin tier of a three-tiered cache for later use (we use S3 as this origin tier). Chunks in the shared storage are named according the their content, ensuring that chunks with the same content have the same name and can be cached once. This scheme, described in detail in Section 3, allows efficient deduplication of chunk content in storage and cache layers without requiring a central directory or index of chunks.

Each fixed-size chunk is 512KiB. Smaller chunks lead to better deduplication by minimizing false-sharing, and can accelerate loading for workloads with highly random access patterns. Larger chunks reduce metadata size, reduce the number of requests needed to load data (hence improving throughput), and provide natural read-ahead for sequential workloads. The optimal value will change over time as the system evolves, and we expect that future iterations of the system may choose a different chunk size as our understanding of how customers use the system evolves.

2.1 Per-MicroVM Snapshot Loading

Once chunks are created, the system needs to be able to access the data they require from the chunks that contain that data. As shown in Figure 4, we added two new components to support this loading:

- A per-function *local agent* which presents a block device to the per-function Firecracker hypervisor (via FUSE), which is then forwarded using the existing virtio interface into the guest, where it is mounted by the guest kernel.
- A per-worker *local cache* which caches chunks of data that are frequently used on the worker, and interacts with the remote cache (see Section 4 for details)

When a new Lambda function is started on a worker, the Micro Manager creates a new *local agent*, and a new Firecracker MicroVM which contains two *virtio* block devices: a

root device which is the same for all MicroVMs, and a block device backed by the FUSE filesystem exposed by the *local agent*. The MicroVM boots, starts some supervisory components, and then starts executing the customer code in the container image. Each IO that this code performs (unless it can be served from the page cache kept by the guest kernel) turns into a *virtio-blk* request, which is then processed by Firecracker, and handed off to the local agent.

The local agent handles reads by reading directly from the local cache, if the chunk that contains the requested offset is already present there. If not, the relevant chunk is fetched from the tiered cache, as described in Section 4. The local agent handles write by writing them to block overlay, backed by encrypted storage on the worker. A bitmap is maintained at page granularity, indicating whether data should be read from the overlay, or from the backing container image. The page granularity of the bitmap requires a read-modify-write for writes from the guest which don't cover an entire page.

This page-level copy-on-write approach allows the MicroVM guest to handle both reads and writes, while keeping the data in the local cache (and all other caching tiers) immutable, allowing it to be shared across multiple guests.

3 Deduplication Without Trust

Base container images, such as the official Docker *alpine*, *ubuntu*, and *nodejs* are extremely widely used: each boasts over a billion aggregate downloads from the popular DockerHub container repository[1]. Starting from one of these base images, and customizing it to the special needs of the application, is a common way to create new container images. When a popular base image is used, the deterministic flattening process described in Section 2 produces unique chunks for the customized parts, and chunks for the common parts that are identical to those produced for other images with the same base. These shared chunks create a significant opportunity for deduplication: if only a single copy of these chunks is stored, less data movement is needed, less storage is consumed, and caches are more effective.

Approximately 80% of newly uploaded Lambda functions result in zero unique chunks, and are just re-uploads of images that had been uploaded in the past. This appears to be primarily driven by automated testing and deployment (CI/CD) systems. Of the remaining 20% of functions that create at least one unique chunk (and therefore aren't just trivial re-uploads), the mean upload contains 4.3% unique chunks, and the median 2.5% unique chunks. Trivial all-zero chunks are not included in these numbers: they are excluded entirely from images at creation time.

Figure 5 shows the distribution of deduplication effectiveness, for the top quartile (by image size) and remainder of the population. This breakdown shows that the majority of functions of all sizes are heavily deduped, and a significant tail where deduplication is not as effective. While large functions are still effectively deduplicated, they have a smaller tail of unique chunks. This data clearly suggests that deduplication is worth the complexity, reducing storage by as much as 23x, and improving effectiveness of the cache tiers (how much cache effectiveness is improved depends on the correlation between probability of deduplication and frequency of access). While the 80% of functions with no unique chunks aren't statistically interesting, deduplicating these has a large practical benefit, including reducing storage costs by another 5x, and boosting cache effectiveness.

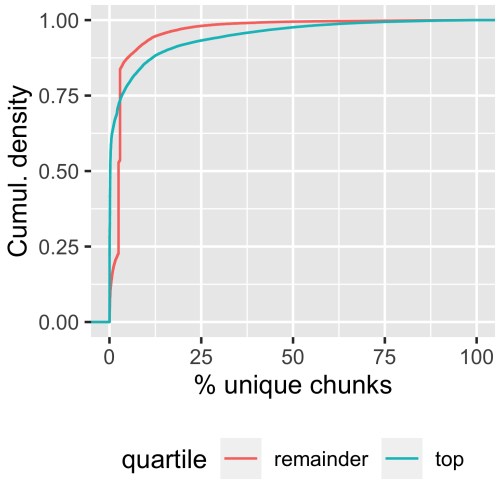


Figure 5: Empirical CDF of deduplication effectiveness at chunk creation time, among functions that aren't trivial re-uploads.

3.1 Convergent Encryption

Deduplication of plaintexts is relatively straightforward. Venti [30], dating back to 2002, used a hash of block contents and a separate index to deduplicate blocks. Introducing encryption, however, significantly complicates deduplication. As Storer, et al [33] write:

> Unfortunately, deduplication exploits identical content, while encryption attempts to make all content appear random; the same content encrypted with two different keys results in very different ciphertext. Thus, combining the space efficiency of deduplication with the secrecy aspects of encryption is problematic.

One solution is to have a shared key, or keys, that can be used to decrypt shared blocks, but this either introduces single keys that can access a large number of blocks, or a significant key management problem. Perhaps the hardest problem is

[1] statistics from https://hub.docker.com/, accessed July 2022

minimizing trust. While AWS Lambda runs user code with strong isolation [3], we still wish to restrict each Lambda worker host to only being able to access the data it needs for the functions that have been sent to it.

The authors of Farsite [2, 11] developed *convergent encryption* as a solution to this problem. A cryptographic hash of each block (in the case of Farsite a file block, in our case a chunk of a flattened container image) is used to deterministically derive a cryptographic key that is used for encryptng the block. We follow this same scheme, but mix additional metadata into the key derivation (as described in Section 3.3).

The flattening process described in Section 2 takes each chunk, derives a key from it by computing its SHA256 digest, and then encrypts the block using AES-CTR (with the derived key). Here, AES-CTR is used with a deterministic (all zero) IV, ensuring that the same ciphertext always leads to the same plaintext. Using a deterministic IV in this context is safe, because due to the collision resistance of SHA256, a *key, IV* pair is only used on for one plaintext block [12]. A manifest of chunks is then created, containing the offset, unique key, and SHA256 hash of each chunk[2]. The manifest is then encrypted, using AES-GCM, using a unique per-customer key managed by AWS Key Management Service (AWS KMS). Chunks are then named based on a function of the hash of their ciphertext, and uploaded to the backing store (AWS S3) using that name if no chunk of that name already exists.

In our scheme, we do not encrypt the entire manifest with the customer's unique key. Instead, only the key table (the keys of each encrypted chunk) is encrypted, and the whole document is authenticated (i.e. included in the calculation of the AES-GCM tag as additional data). This allows the garbage collection process to access the list of chunks in the manifest, while having no access to the chunk keys. The size of manifests, stored in an efficient binary format, is negligible: less than 3MiB for a 16GiB container image, or 0.02% overhead.

This approach provides a number of desirable properties:

- Data can be deduplicated with no sharing of keys: the keys to decrypt the customer's manifest are unique to that customer, and access to them (via AWS KMS) is only provided to the workers that that particular customer's functions are placed on.
- Data can be deduplicated with no coordination or special access provided to the flattening process. Flattening processes operate independently, and the only special operation they need is "upload this file to storage if it doesn't already exist".
- The scheme provides strong end-to-end integrity protection for chunks. Workers check the chunks they download against the MAC in the manifest, ensuring that modified ciphertexts can be detected and rejected.

[2]It may appear attractive to use an AEAD mode like AES-GCM rather than the more expensive SHA256 in this application, but these modes do not commonly provide collision resistance against attackers who know the data key [10], an important property in our security scheme.

3.2 Compression

Our system does not compress chunk plaintexts prior to encryption. This is for two reasons. First, given the network bandwidth available to our caches and workers the additional latency of decompression, and difficulty of allowing random access to compressed data, makes the latency benefit of compression marginal. Second, compression before encryption allows potential attackers to infer plaintext contents from compressed sizes, a *compression side channel*. This risk, and the relatively small expected benefit, means that we decided not to implement compression (beyond trivial elision of all-zero chunks).

3.3 Limiting Blast Radius

While deduplication has value in cost and cache performance, it also adds some risks. Some popular chunks are widely referenced, meaning that anything that causes access to those chunks to break or become slow, also has a very wide impact on the system. Risks include partial (gray) failures of cache nodes, operational issues that cause unavailability of data, bugs in garbage collection, or corruption of data in the cache hierarchy. Highly popular chunks also cause hot-spotting in distributed storage. While our cryptographic scheme detects corruption and will prevent readers from seeing corrupt data, it does not correct it, and so corrupted data will become unavailable.

To solve this problem, we include a varying *salt* in the key derivation step of our convergent encryption scheme. This salt value can vary in time, with chunk popularity, and with infrastructure placement (such as using different salts in different availability zones or datacenters). Otherwise-identical chunks with different salt values will end up with different keys, and therefore difference ciphertexts, and will not deduplicate against each other. By controlling the frequency with which the salt is rotated, we can continuously trade off deduplication efficiency with blast radius. Salt allows us to encapsulate the control of deduplication entirely within the chunk creation layer, without any other component needing to be aware of its decisions. Salt rotation is an operational concern, and is not needed for the security of the deduplication scheme.

3.4 Garbage Collection

A key challenge of any distributed deduplication scheme is garbage collection: removing data from the backing store when it is no longer actively referenced. Garbage collecting the wrong chunk could cause wide impact across multiple customers. Our deduplication scheme does not maintain a central

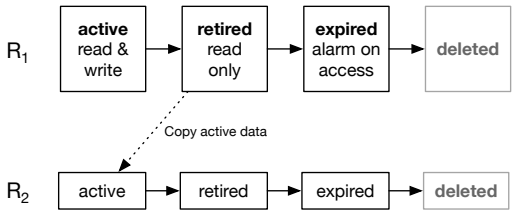


Figure 6: Lifecycle of data chunks used by the generational garbage collector.

directory of chunk references or manifests, making exact reference counting infeasible. Past experience with distributed garbage collection has taught us that the problem is both complex (because the tree of chunk references is changing dynamically) and uniquely risky (because it is the one place in our system where we delete customer data). The approach we took to garbage collection is based on this experience.

Our approach to garbage collection is based on the concept of *roots*. A *root* is a self-contained manifest and chunk namespace, analogous to the *root*s used in traditional garbage collection algorithms. Unlike traditional GC roots, in our system we periodically create new roots (which then get all new data), and retire old roots (after moving any still-needed data into a fresh root).When a customer's container image is converted, the manifest and set of chunks are placed in an *active* root, for example R_1. An active root handles both reads and writes of data. Periodically, a new root R_2 is created and becomes active, while root R_1 enters a *retired* state at which point it only serves reads of data. While R_1 is retired, any manifest that is still referenced in R_1 is migrated, along with any chunks it references, to R_2. Over time the manifests and chunks in R_1 that are in active use will be migrated to R_2, allowing R_1 to be safely deleted. This process is repeated: R_2 is retired and R_3 becomes the active root and so on. Figure 6 shows this lifecycle. Moving chunks along with their manifest ensures that if a manifest exists in root R, then all the chunks it references do to. A unique identifier for the currently active root is also included in the deduplication salt (Section 3.3), ensuring that newly-created chunks in the active root are not shared with previous roots.

Instead of deleting roots immediately after data migration is complete, we put them into an *expired* state. In this state, data is still allowed to be read, but any attempt to access data leads to an alarm. These alarms both engage an operator and automatically stop further deletion of data. This approach allows us to robustly detect garbage collection issues (especially incomplete copying) in production, and quickly and automatically stop any data from being deleted. While this mechanism is inexact (data could be accessed after the period the root is *expired*), it provides a valuable additional layer of protection against data loss. While software bugs are rare, and we test garbage collection changes carefully, multiple layers of protection against customer data loss are critical in any distributed storage system.

Having data in multiple roots does drive up storage costs, however that additional cost is palatable for Lambda as customers often update their functions and a large majority of data is never migrated to a new root. The system is also capable of having multiple roots active simultaneously, which reduces the blast radius of bugs and provides the ability to roll out new garbage collection changes and algorithms to a subset of manifests and their chunks.

4 Tiered Caching

When workers don't have chunks in their local cache, they attempt to pull them from a remote availability-zone-level (AZ-level) shared cache (as shown in Figure 3). If chunks aren't in this cache, workers download them from S3, and upload them into the cache. This AZ-level cache is a custom implementation of a fairly standard design: chunks are fetched over HTTP2, data storage is two-tiered with an in-memory tier for hot chunks and a flash tier for colder chunks, and eviction is LRU-k [29] (a scan-resistant variant of Least Recently Used). Chunks are distributed to the AZ-level cache using a variant of a consistent hashing [19] scheme, with optimizations to improved load spreading (similar to the approach of Chen et al [7]). The caching tier improves fetch performance considerably: from the worker's perspective, a hit on the AZ-level cache takes a median time of 550μs, versus 36ms for a fetch from the origin in S3 (99.9th percentile 3.7ms versus 175ms).

Figure 7 shows the effectiveness of these three cache tiers. Over a week of production usage in one large AWS region, a median of 67% of chunks were loaded from the on-worker cache, 32% from the AZ-level distributed cache, and the remaining 0.06% from the backing store.

The per-worker cache has a median hit rate of 67%, and a 10th percentile low hit rate over the week in question of 65%. The in-AZ cache is even more effective, with a median hit rate of 99.9% and 10th percentile low hit rate over the week of 99.4%. Figure 8 shows the empirical CDF of the hit rate of the in-AZ cache over the week, measured in one-minute buckets across one at-scale production availability zone. The left tail of the distribution is associated with large spikes in traffic to newly created functions. We are evaluating priming the in-AZ caches during the chunk creation process to flatten this left tail and further improve hit rates, primarily with the goal of reducing load-time latency for new functions.

4.1 Optimizing for Tail Latency

While data in the AZ-level cache is not required to be durable (durability is ensured using S3 as the origin), a simple unreplicated cache scheme (where each object is stored in a single node) didn't meet our needs for three reasons.

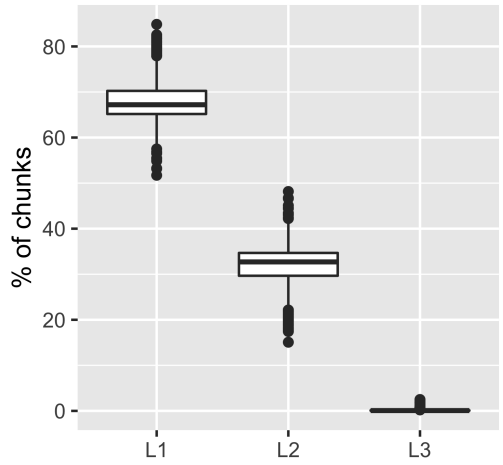


Figure 7: One week of hit rates on each of the cache tiers: on-worker (L1), distributed in-AZ (L2), and backing store (L3)

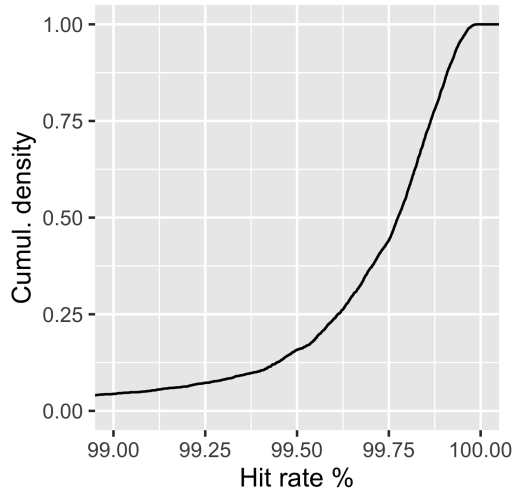


Figure 8: Empirical CDF of in-AZ cache hit rate

Tail latency A single slow cache server can cause widespread impact. Slowness could be caused by congestion at the host or in the network, or by partial hardware or software failure.

Hit Rate Drops Having each item cached in a single server means that the hit rate drops if that server fails, or is taken down for deployment.

Throughput Bounds Having each item cached in a single server means that the bandwidth available to fetch the object is bounded by a single server's bandwidth.

Of these, tail latency is the largest practical concern. Our experience operating these types of systems suggests that debugging slowness and partial failure is much harder than debugging outright failure. Even if this slowness is only in the long tail, it still matters in production because each container start needs to fetch a large number of chunks. For example, a start which fetches 1000 chunks will experience the 99.9th percentile tail latency of the cache on 63% of tasks. The difference is material: in one deployment of the cache we observe a median client-measured latency of 500μs, and a 99.9th percentile latency of 4ms.

Replication, combined with redundant requests is a well-established [13, 37, 39] technique to drive down tail latency, and would also solve our throughput and hit-rate problems. Unfortunately, replication increases costs proportionally to the replication factor, an important concern in a primarily in-memory cache. Instead, we chose erasure coding, following a similar scheme to EC-Cache [31]. Erasure coding is not widely used in caches, but provides compelling solutions for all three of our concerns. When a worker misses the cache, it fetches the chunk it needs from the origin, then uploads erasure-coded stripes of that chunk into the cache. When a worker needs to fetch a chunk, it requests more stripes than are strictly needed to reconstruct the chunk, and then reconstructs the chunk as soon as enough stripes are returned. Our current production deployment uses a *4 of 5* code, achieving 25% storage overhead, and a 25% increase in request rate in exchange for a significant decrease in tail latency. Figure 9 compares the empirical latency CDF of the *4 of 5* code versus a hypothetical *4 of 4* scheme using latency measurements from one deployment of our production system.

This scheme prevents any drop in hit rate from occurring when cache nodes fail, or are taken down for deployment. A common approach in similar systems is to use retries to hide the effects of deployments and failed nodes, an approach which is known to lead to metastable failure modes in large systems [5, 17]. Erasure coding allows us to achieve a similar level of resiliency while performing the same amount of work in success and failure cases (a design philosophy we call *constant work* [23]).

4.2 Stability and Metastability

Caches with high hit rates, such as ours, are desirable from a latency and efficiency perspective, but have a hidden downside. If the cache becomes empty (such as due to power loss or operational issue), or the hit rate suddenly drops (such as due to a change in customer behavior), the downstream services can see significantly more traffic than they are used to. In the case of our cache, with an end-to-end hit rate typically exceeding 99.8%, this downstream traffic increase could be up to 500 times normal. S3 is an extremely scalable backing store, and can tolerate the full uncached load. However, the increased latency leads to higher concurrency demand from customer's applications (due to Little's Law [21]), and therefore higher demand for new Lambda slots, increasing load and changing

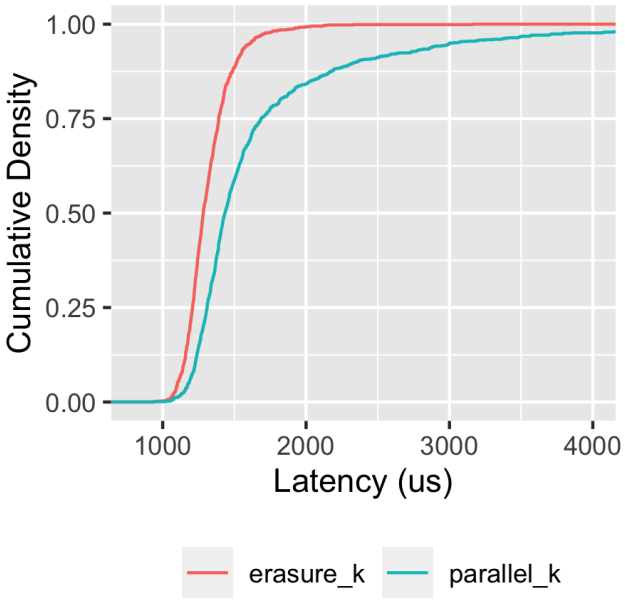


Figure 9: Comparative empirical CDFs of client-side latency of 4-of-4 parallel cache load, versus 4-of-5 erasure coded cache load.

the size and composition of the system's working set. This can lead to metastable behavior [5, 6, 17], where the system isn't able to refill the cache when it is empty[3].

We have built mitigations for this risk into higher layers of Lambda. Primarily, the system is designed to be concurrency-limited. When container starts slow down and the number of concurrent tasks exceeds this limit, new starts are rejected until in-flight ones complete. We also actively test the system's ability to *cold start* from an empty cache at the maximum concurrency. This testing allows us to be confident that the system is able to restart from a cold cache, or tolerate workload changes that significantly reduce hit rate.

4.3 Cache Eviction and Sizing

Traditional cache replacement policies like Least Recently Used (LRU) and First In First Out (FIFO) are simple and easy to implement, but have a significant downside for this application: a lack of *scan resistance*. In our case, this means that a large number of infrequently used functions starting up can replace all the hot entries in the cache with recently-used entries belonging to those functions, dropping cache hit rates for more frequently-used entries, and filling the cache with entries that will never be read again. This happens periodically in our environment, driven by weekly, daily, and hourly spikes of periodic *cron job* functions. These functions are large in number, but each runs at a low scale (typically only using one sandbox), making caching their chunks relatively unimportant. To avoid the hit-rate drops caused by this periodic work, we use the LRU-k [29] eviction algorithm, which tracks the last k times an item in the cache was used, rather than only the most recent time.

Eviction and hit rates are also related to the size of our local and AZ-level caches. Following the logic of Gray and Putzolu's classic *Five Minute Rule* [14], the minimum desirable cache size is the one that makes the cost of cache retention equal to the cost of fetching chunks from S3. However, because our cache is not only aimed at reducing costs but also improving customer-observed latency, we also set a hit rate goal and increase the cache size if we fall below that goal. The total cache size, then, is the larger of the size needed to achieve our hit rate goal, and the size needed to optimize costs.

5 Implementation and Production Experience

We built the local agent (the FUSE implementation that backs the sparse block device for each MicroVM), the worker-local cache, and the remote cache server in the Rust programming language. We used the *tokio* runtime, and *reqwest* and *hyper* for HTTP. At the time we started this project, the invoke path of AWS Lambda includes components written in Java, Go, C, and Rust. We chose Rust because of our good experiences with the Rust components we had built in the past, especially around performance and stability, and have again been happy with our choice of Rust, encountering no major production bugs in the libraries we chose. We were also attracted to Rust because of the successes other AWS teams (such as the Amazon S3 team [4]) have had applying formal methods to verify code correctness in Rust, even with non-expert programmers.

One interesting stumbling block with Rust (version 1.46.0, current at the time of implementation) is brittle optimization, especially autovectorization, of hotspots. Unsurprisingly, we found that the parity calculations we use for erasure coding are nearly 5x faster when performed 64 bytes at a time (with AVX512) or 32 bytes at a time (with AVX or NEON) than when performed 8 bytes at a time, and 10x faster than when performed byte-at-a-time. Unfortunately, the naive Rust loop emitted the byte-at-a-time code (as shown in Listing 1), despite the compiler being capable of autovectorization. Small changes to the code would change autovectorization behavior, even changes outside the function of interest. Reluctant to move to assembly for this code, we finally settled on the code in Listing 2, which robustly emits appropriately unrolled

[3] Related effects have been observed in computer systems since at least the 1960s. In the 1968 paper 'The Working Set Model for Program Behavior' [9], Peter J Denning observed a similar effect in paging systems:

> This can create a self-intensifying crisis. Programs, deprived of still-needed pages, generate a plethora of page faults; the resulting traffic of returning pages displaces still other useful pages, leading to more page faults, and so on.

AVX, AVX512, or vectorized ARM code depending on the target platform. Seemingly small changes to this function (such as removing the *assert*, changing any of the assignments, or allowing it be inlined) cause autovectorization to be disabled. This is a small issue with Rust, and one that we expect to be improved in future compiler versions.

Listing 1 Naive byte-by-byte x86 assembly code as emitted by the Rust compiler for straightforward loop implementation (with annotations by *perf* showing percent of runtime). Note significant missed opportunities for optimizations like vectorization and loop unrolling.

```
  0.08 |350:   cmp    %rax,%rsi
       |     ↓ jae    3f4
 49.18 |       movzbl (%rdi,%rsi,1),%ebx
  0.13 |       xor    %bl,(%rcx,%rsi,1)
 50.52 |       lea    0x1(%rsi),%rbp
  0.08 |       mov    %rbp,%rsi
       |       cmp    %rax,%rbp
       |     ↑ jb     350
```

Listing 2 Implementation of parity calculation in Rust, showing extra lines needed for reliable autovectorization.

```
#[inline(never)]
fn parity(target: &mut [u8], source: &[u8]) {
    assert_eq!(source.len(), target.len());
    let len = target.len();
    let _ = target[len-1];
    let _ = source[len-1];

    for i in 0..len {
        target[i] ^= source[i];
    }
}
```

On the other hand, the Rust ecosystem's support for build-time microbenchmarks (such as with the *criterion* crate) makes it fast and easy to iterate on this type of performance work, and even assert at build time that autovectorization has succeeded (effectively stopping regressions from entering production). This is a significant boon in a cloud environment, where performance regressions can cause production outages, and performance is tied to both cost and carbon efficiency.

5.1 Latency and Multimodality

As with any storage system, performance was an important goal for the design and implementation of our snapshot chunk loading system. While throughput, CPU efficiency, and other bandwidth measures contribute to the cost of running the system, its scale-out nature make latency and scalability the most important factors of performance. The local agent and on-worker caches trivially scale out, due to the fact that they do not communicate off their worker, except in interacting with S3 (to pull chunks from the origin), and the L2 AZ-level cache.

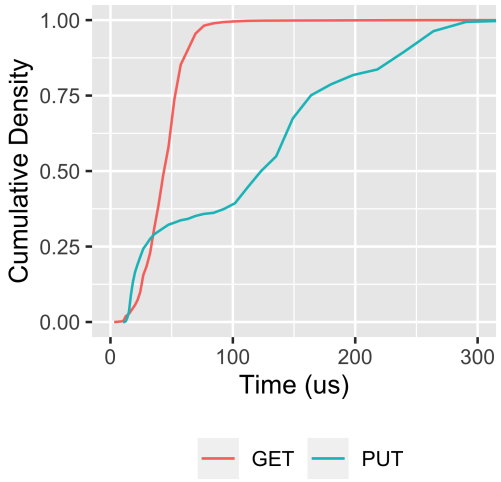


Figure 10: Empirical CDF of server-side measured latency of the L2 cache server

Figure 10 shows the latency for GETs and PUTs on this cache, measured from the server side, across all of the cache nodes in a production deployment over the course of one week. Each GET or PUT is of a 512kB chunk. As discussed in Section 4, the L2 cache is a flash-based cache with a significant local memory tier (about 10% of cache size). GET latency is very consistent, with a median of below 50μs. PUT latency is less consistent, with some multi-modality apparently caused by writeback behavior on the cache host. Despite this multi-modality, performance is still excellent, with a median latency of 125μs, a 99th percentile latency below 300μs, and a 99.99th percentile of 413μs[4]. When building this cache server, we chose HTTP2 as a wire protocol for convenience with the intention of replacing it with an efficient binary protocol later. In production, we've found the overhead of HTTP (implemented with *hyper* and *reqwest*) so low that we have not yet been motivated to replace the protocol.

Figure 11 shows the end-to-end latency for returning a read from the perspective of the local agent (that is the FUSE implementation). This doesn't show the end-to-end IO latency experienced by guests, because it's from the perspective of the worker and does not include the (significant) hit rate on the page cache maintained by the MicroVM guest's kernel, and read-ahead performed by the guest to populate that cache. Like the L2 server latency, this end-to-end latency shows sig-

[4]Having a 99.99th percentile at less than 4x the median is a very desirable property, and difficult to achieve with garbage collected languages like Java and Go

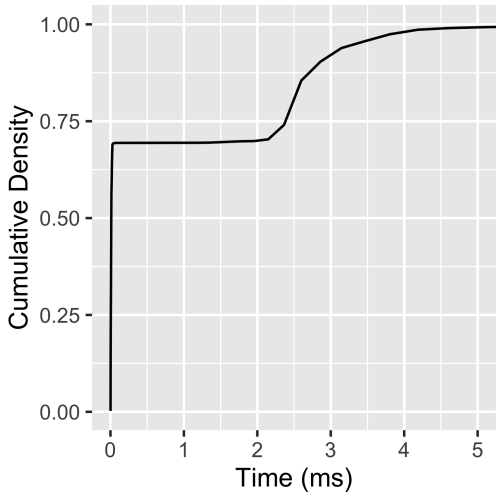


Figure 11: Empirical CDF of end-to-end read latency observed at the local agent (FUSE implementation).

nificant multi-modality: a mode below 100μs which represent local cache hits, a mode around 2.75ms which represent L2 hits (and the subsequent work like decryption), and mode (trimmed from the graph) showing rare fetches from the origin (see Figure 7 for the relative frequencies of these modes). We are working on an optimized cryptographic scheme which reduces the latency of decryption.

Multi-modality like this is the norm in storage systems, but presents a few practical challenges to operators. First, as discussed in Section 7 a small change in the relative frequencies of each mode can significantly change the mean latency observed by clients (and so change the concurrency and throughput of the system). Second, latency percentiles and trimmed means are the summary statistics most commonly used by operators at AWS, and they tend to obscure multi-modality. Plots like empirical CDFs (eCDFs, as presented here) can be valuable, but don't show change-over-time as time series of summary statistics do. We have experimented with heat maps, day-over-day eCDFs, and others, but have yet to find a succinct way to present these data to operators. Third, multi-modality makes the decision of where to spend optimization resources more complex. Which mode should the team work to improve? Or should they work to reduce the relative frequencies of higher modes?

5.2 Production experience with FUSE

Our experiences with FUSE match those reported by Vangoor et al [36], showing relatively little throughput overhead when well tuned. However, we have found that the choice to use FUSE to present a file which is then subsequently used as a block device by Firecracker's virtio-blk implementation, has introduced significant overhead. When an application running in a MicroVM reads a new chunk, control is passed to the guest kernel, then Firecracker, then the host kernel's FUSE layer, then the local agent, before flowing back through the same path. This introduces context switch overhead, but more importantly requires four different threads to be scheduled by the host kernel's scheduler. This introduces inefficiency in steady state, and significant jitter under load. We are moving away from FUSE for this application, primarily due to this effect. Our new implementation uses *userfaultfd* and *mmap*, removing two layers from the architecture.

We don't regret starting with FUSE. It provided a convenient interface, a clear security and operational isolation story, and allowed a team without deep systems-level programming experience to build an acceptably high performance system.

6 Related work

Mirroring the rise in popularity of serverless and containers accelerated container loading has been a highly active area of research, and industry implementation, over the last decade. Before that, accelerating VM loading through faster disc image movement was an active area of research. For example, Frisbee [16] in 2003. Amazon EC2 has taken advantage of common data to accelerate VM image loading, through tracking lineage of EBS snapshot chunks [28], since 2009. With Slacker [15] Harter et al studied access patterns in container loading, and presented a system which takes advantage of these patterns by performing layer-level lazy loading. Starlight [8] takes a fairly similar filesystem-orientated approach, optimized for loading at the edge where minimizing round-trips to the datacenter is a significant contributor to performance. eStargz [35] extends common container image formats to make lazy loading at the layer level more efficient, building on the approach of Google's CRFS.

DADI [20] uses a block-level approach fairly similar to our own, but with a peer-to-peer approach rather than a dedicated cache layer, and without the ability to deduplicate as widely as our system is able to. FaaSNet [38] approaches a similar problem to the one we were solving, but works on the layer level (rather than flattening images as we do), and does not appear to perform deduplication. Cntr [34] and Yolo [26] take the approach of breaking down container images into different classes of data, some needed urgently on start up and some likely to be accessed less urgently. This explicit approach may be more efficient than the simple block-based approach, but also requires a deeper introspection of the contents of the container. Wharf [41] and CFS [22] take the distributed filesystem approach, showing that can significantly improve loading performance at the cost of increased coordination between containers.

Accelerating storage performance and loading with deduplication has an even longer history, for example in 2001 with Muthitacharoen et al [25] and 2002 with Venti [30], and Farsite [2].

7 Conclusion

We present AWS Lambda's solution for accelerated loading of container images, and approach that combines deduplication, erasure coding, tiered caching, userspace filesystems, and convergent encryption. We have operated this system for several years, and are extending its use into other areas of AWS. While our solution on the surface appears to have a lot of moving parts, it is optimized for what we believe to be the realities of building massive scale cloud systems: failures are frequent, failures are often partial and complex, and security is the top priority.

7.1 Broader Lessons and Future Work

While Lambda's snapshot loading infrastructure is a specialized system for a rather specialized application, we believe that there are some broader lessons from our experiences that apply to the systems community as a whole.

- Containers are most popularly used by Lambda customers as "static linking in the large" dependency closures. Customers want to build, test, and deploy a function with all its dependencies in one atomic unit, but traditional static linking is either unavailable or inconvenient. However, containers are also highly inefficient in this context, necessitating the deduplication and sparse loading we describe here. We believe that there is a significant need for a lighter-weight dependency closure mechanism, which comes closer to traditional static linking in the size of the artifacts that it creates.

- Caches reduce costs, improve latency, and reduce load on durable storage, and are a critical component of nearly any stateful system. However, they also introduce risks such as metastable failures (due to unexpectedly empty caches, or sudden shifts in workloads), and challenges for users like multi-modal latency distributions. While work such Yang et al [40], and Huang et al [17] have made steps towards deeply understanding these effects, we believe that significantly more work is needed to understand the dynamic behaviors of caching in large systems, and to develop patterns to mitigate the risks of caches.

- MicroVMs provide an isolation mechanism which is nearly as lightweight as containers, or even processes [3, 24], while providing additional interfaces for plugging in both local and distributed operating system logic. MicroVMs provide a powerful new tool in the operating system researcher's or builder's toolbox. We believe that operating system support for virtualization, and virtualization support for applications, operating systems, and databases are ripe areas of research which are not yet receiving sufficient attention.

Our future work is focused on optimizing the system further for cost, performance, and especially customer-experienced cold-start latency. This same system is used in Lambda SnapStart, a feature of AWS Lambda which reduces cold-start latency using memory snapshots, to store and load memory snapshot contents. That use-case is especially latency sensitive, motivating significant investments in both average case and tail latency. We expect this work to include optimizing cache retention and data placement policies, optimizing client and server performance, and completing the migration from FUSE to *userfaultfd*.

Acknowledgements

Any system of this size requires a team to build and operate, and in this case we're deeply thankful to the AWS Lambda team for their work and contributions. Holly Mesrobian, David R. Richardson, Ajay Nair, and David Nasi were instrumental in supporting this work. Shay Gueron, Osman Surkatty, and Derek Manwaring helped ground our cryptographic ambitions, and provided valuable feedback.

References

[1] Oci image format specification. Accessed: 2022-04-15. URL: https://github.com/opencontainers/image-spec.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, Boston, MA, December 2002. USENIX Association. URL: https://www.usenix.org/conference/osdi-02/farsite-federated-available-and-reliable-storage-incompletely-trusted-environment.

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

[4] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21,

page 836–850, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483540.

[5] 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. doi:10.1145/3458336.3465286.

[6] Marc Brooker. Some risks of coordinating only sometimes. In *High Performance Transaction Systems 2019 (HPTS'19)*, November 2019.

[7] John Chen, Ben Coleman, and Anshumali Shrivastava. Revisiting consistent hashing with bounded loads, 2019. URL: https://arxiv.org/abs/1908.08762, doi:10.48550/ARXIV.1908.08762.

[8] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. Starlight: Fast container provisioning on the edge and over the WAN. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 35–50, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/chen-jun-lin.

[9] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, may 1968. doi:10.1145/363095.363141.

[10] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I*, page 155–186, Berlin, Heidelberg, 2018. Springer-Verlag. doi:10.1007/978-3-319-96884-1_6.

[11] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, page 617, USA, 2002. IEEE Computer Society.

[12] Morris J Dworkin. *NIST SP 800-38D. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac*. National Institute of Standards & Technology, 2007.

[13] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. Reducing latency via redundant requests: Exact analysis. *SIGMETRICS Perform. Eval. Rev.*, 43(1):347–360, jun 2015. doi:10.1145/2796314.2745873.

[14] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 395–398, 1987.

[15] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, February 2016. USENIX Association. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter.

[16] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, scalable disk imaging with frisbee. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, San Antonio, TX, June 2003. USENIX Association. URL: https://www.usenix.org/conference/2003-usenix-annual-technical-conference/fast-scalable-disk-imaging-frisbee.

[17] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/osdi22/presentation/huang-lexiang.

[18] Vlad Ionescu. Scaling containers on aws in 2022. Accessed: 2022-04-15. URL: https://www.vladionescu.me/posts/scaling-containers-on-aws-in-2022/.

[19] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258533.258660.

[20] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI: Block-Level image service for agile and elastic application deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 727–740. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/li-huiba.

[21] John DC Little. A proof for the queuing formula: L= λ w. *Operations research*, 9(3):383–387, 1961.

[22] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. Cfs: A distributed file system for large scale container platforms. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1729–1742, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3299869.3314046.

[23] Colm MacCárthaigh. Reliability, constant work, and a good cup of coffee, 2020. URL: https://aws.amazon.com/builders-library/reliability-and-constant-work/.

[24] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132763.

[25] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 174–187, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/502034.502052.

[26] Thuy Linh Nguyen, Ramon Nou, and Adrien Lebre. Yolo: Speeding up vm and docker boot time by reducing i/o operations. In *European Conference on Parallel Processing*, pages 273–287. Springer, 2019.

[27] OASIS. Virtual i/o device (virtio) version 1.0, March 2016.

[28] Marc Olson and Prarthana Karmakar. Amazon ebs under the hood: A tech deep dive, December 2021. URL: https://www.youtube.com/watch?v=kaWzAEVZ6k8.

[29] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[30] Sean Quinlan and Sean Dorward. Venti: A new approach to archival data storage. In *Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. USENIX Association. URL: https://www.usenix.org/conference/fast-02/venti-new-approach-archival-data-storage.

[31] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-Balanced, Low-Latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, Savannah, GA, November 2016. USENIX Association. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/rashmi.

[32] Rusty Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. URL: http://doi.acm.org/10.1145/1400097.1400108, doi:10.1145/1400097.1400108.

[33] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS '08, page 1–10, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1456469.1456471.

[34] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/atc18/presentation/thalheim.

[35] Kohei Tokunaga. Startup containers in lightning speed with lazy image distribution on containerd. Accessed: 2022-04-15. URL: https://medium.com/nttlabs/startup-containers-in-lightning-speed-with-lazy-image-distribution-on-containerd-243d94522361.

[36] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, February 2017. USENIX Association. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor.

[37] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, page 283–294, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2535372.2535392.

[38] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless

container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021. URL: https://www.usenix.org/conference/atc21/presentation/wang-ao.

[39] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 543–557, Oakland, CA, May 2015. USENIX Association. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/wu.

[40] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/yang.

[41] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 174–185, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3267809.3267836.

Decentralized and Stateful Serverless Computing on the Internet Computer Blockchain

Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, Rostislav Rumenov, Dimitris Sarlis, Alin Sinpalean, Alexandru Uta, Bogdan Warinschi, Alexandra Zapuc
DFINITY, Zurich

Abstract

The Internet Computer (IC) is a fast and efficient decentralized blockchain-based platform for the execution of general-purpose applications in the form of smart contracts. In other words, the IC service is the antithesis of current serverless computing. Instead of ephemeral, stateless functions operated by a single entity, the IC offers decentralized stateful serverless computation over untrusted, independent datacenters. Developers deploy stateful *canisters* that serve calls either to end-users or other canisters. The IC programming model is similar to serverless clouds, with applications written in modern languages such as Rust or Python, yet simpler: state is maintained automatically, without developer intervention.

In this paper, we identify and address significant systems challenges to enable efficient decentralized stateful serverless computation: scalability, stateful execution through orthogonal persistence, and deterministic scheduling. We describe the design of the IC and characterize its operational data gathered over the past 1.5 years, and its performance.

1 Introduction

Recently, the technological advances in blockchain [29], cryptography [27] and consensus protocols [5, 9, 24] have enabled more and more efficient execution of decentralized Web3 [56] applications and smart contracts. Platforms that service such applications are larger than ever [42], consisting of thousands of nodes, processing billions of requests, storing large quantities of data and connecting many users. Currently, the research community lacks a clear understanding of the operational data of such large-scale platforms, their challenges and performance, beyond testnet deployments with synthetic workloads and failure patterns. In this article, we introduce the Internet Computer (IC), its design, several of its systems challenges and real-world operational performance data.

The IC is a decentralized platform for the execution of general-purpose decentralized applications (dapps). Listing 1 shows an example for such a dapp. In current serverless offerings, this application would not work without an external service, as functions are stateless. Instead, the IC enables `decentralized and stateful serverless` computing. The IC protocol [53] runs on globally distributed servers in independent datacenters. It is highly scalable and efficient in executing applications. The main goals of the IC are decentralization, security and performance.

```
use ic_cdk_macros::{query, update};
use std::{cell::RefCell, collections::HashMap};

thread_local! {
    static STORE: RefCell<HashMap<String, u64>> = RefCell::default();
}
#[update]
fn insert(key: String, value: u64) {
    STORE.with(|store| store.borrow_mut().insert(key, value));
}
#[query]
fn lookup(key: String) -> u64 {
    STORE.with(|store| *store.borrow().get(&key).unwrap_or(&0))
}
```

Listing 1: Functional key-value store canister. The update call adds a key value pair; the query call gets values by keys. State is stored on the canister heap and persisted transparently.

In particular, the IC aims to enable governance and evolution to be controlled by different parties in a trustless and fault-tolerant manner instead of a central entity. The IC must also provide strong integrity and access control guarantees for the apps running on it as well as the users interacting with it in an efficient way. Overcoming these challenges requires novel blockchain technology, cryptography and consensus protocols [9, 27, 53]. Those advances need to be combined with a carefully crafted system design. In this paper, we focus on those systems-related challenges at the application execution layer and we present our solutions and operation data.

Application developers deploy dapps (equivalent to serverless function workflows) on the IC without the cumbersome process of resource management, just like in serverless environments. The dapps interact with each other and with

end-users. Each dapp is composed of *canisters*, the smallest units containing code and data, an immediate equivalent to serverless functions. Such canisters can be combined to build complex and powerful smart contracts. Figure 1 depicts our protocol stack. The IC operates as a large-scale replicated state machine. To achieve wide-range scalability, the IC nodes are partitioned (sharded) [13, 57] into *subnets*, each running its own replicated state machine.

Central to this article is the execution environment, which ensures that the actions implemented by developers are triggered efficiently and deterministically, and that consumed resources are accounted for. For simplicity and portability, applications are programmed in a high-level language such as Rust, but compiled down to *WebAssembly* [28]. Canisters run isolated from one another inside sandboxed processes that execute code running under a WebAssembly virtual machine.

We identified significant execution layer systems challenges that we addressed when designing and building the IC. First, as opposed to serverless environments [49], our applications are long-running and **(C1)** `stateful`. The IC enables this through an efficient mechanism to track modified memory during canister call execution [39]. Coupled with canister statefulness, the IC programming model is inspired by an event-driven, actor-based model, where application programmers implement functionality that responds to messages from users or other canisters. To simplify programmer experience the IC offers `orthogonal persistence` [14, 31] – the system running the canister automatically persists the canister memory state without users taking action toward this goal. Therefore, sending messages to a canister is just like invoking multiple times a serverless function, with the distinction that function state modified by earlier calls is persisted without the programmer explicitly saving data in external services.

Second, similarly to current scalability challenges [30, 50, 58] in serverless computing, our nodes need to run thousands of canisters (or functions) per node. This entails achieving intra-node **(C2)** `scalability`. This is a must to accommodate large subnets with tens of thousands of canisters.

Third, we emphasize **(C3)** `determinism`. Since the IC is a large decentralized replicated state machine, all nodes must transition to the same next state, despite potentially malicious user input, canister code and node behavior. In a first step, this requires strict message *ordering*. Moreover, determinism is necessary in *scheduling* [2, 36, 44] actions that alter the replicated state and, of course, the actual *state changes* must be performed deterministically as well.

Of utmost importance for the IC is ensuring **(C4)** `security` for application developers and end-users. More precisely, the IC design aims to minimize the trust application developers and end users must place in individual entities. Thus, the IC relies on strong integrity, availability and access control guarantees. In particular, only valid messages will be processed and the response can be verified as long as more than two thirds of the nodes are honest. Canisters cannot inspect or change the state of other canisters or other parts of the system. These guarantees are achieved by carefully crafted low-level operating systems and runtime mechanisms as well as through the use of virtualization and sandboxing techniques.

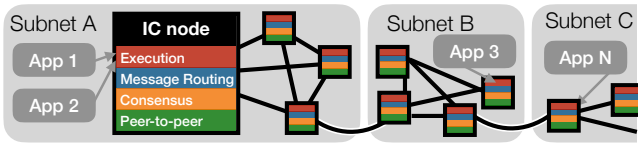


Figure 1: The protocol stack run by the nodes of the Internet Computer. The P2P layer disseminates protocol and user-generated messages. Message validation and ordering is established by the consensus layer. Messages are then routed to the execution environment and trigger efficient deterministic (replicated) computation of the apps deployed on the IC. Apps on different subnets can send messages to each other.

In this paper we show that the years-old mantra saying blockchains are slow and inefficient is coming to an end. Addressing these challenges efficiently means the IC is, to the best of our knowledge, the fastest and most efficient blockchain to date, outpacing the execution speed, transaction and execution costs [16] of other blockchains by orders of magnitude, while having significantly fewer carbon emissions [10]. More importantly, it enables decentralized stateful serverless computing. We therefore omit comparing the IC with other blockchains, but rather compare its performance with native and non-decentralized client-server architectures, whose performance we aim to achieve.

Having operated the IC for more than 1.5 years, we share our experience in designing and building the IC, with an emphasis on its systems challenges. The IC was launched in May 2021. As of January 2023, it hosts over 230,000 canisters for a total state of 2.5 TB (more than twice the size of the Ethereum blockchain) running services ranging from social media to decentralized finance. Our main contributions are:

1. We present the high-level design of the IC (Section 2).
2. We present systems challenges of the IC execution layer, with a focus on the memory subsystem, orthogonal persistence, deterministic scheduling and scalability (Section 3).
3. We showcase the performance of the IC. We introduce high-level operational data, which we open to the public. We present end-to-end application performance and study in-depth the IC performance compared to native applications. We discuss the (performance) implications of decentralization and statefulness (Section 4).

2 The Internet Computer Design

In this section we briefly introduce the IC. For a more comprehensive article explaining protocol aspects in more depth we refer the reader to the IC whitepaper [53].

Motivation. The IC aims to provide efficient multi-tenant, general-purpose, and secure computation in a decentralized

and geo-replicated manner tolerating Byzantine faults, offering developers a modern and easy to use programming model.

Overview. The nodes of the IC run a network of replicated state machines [48], which interact with each other via messages. State machine replication achieves the same output state for a service replicated on multiple machines. Each state machine generates new states by applying *deterministic* state transformations — based on the deterministic execution of the canisters' code provided by the app developers — to the previous state by processing ordered input messages from users and other canisters. The result is a new state and output messages to canisters and users.

Subnets and nodes. The nodes (term used interchangeably with replicas, machines, or servers) of the IC are partitioned into *subnets*, each subnet providing state machine replication for the set of canisters deployed on it.

Each node in a subnet runs all the canisters deployed in that subnet. Subnets can be smaller or larger: we have subnets with 80,000+ canisters and subnets with several hundreds. Most subnets have 13 nodes that are geo-replicated across the Americas, Europe and Asia. For applications in need of improved security, we have higher-replication subnets, spanning up to 40 nodes. A subnet should continue to function even if some replicas are faulty. The replicas running the IC protocol are hosted on servers in geographically distributed, independently operated data centers, bolstering security and decentralization.

Currently, the IC nodes are homogeneous. Homogeneity is important for system parts that are executing code running in the replicated state machine, as otherwise, speed may be reduced due to too many slow nodes. However, functionality outside of the replicated state machine, such as for executing non-replicated query calls can be scheduled proportionally to each machine's resource availability.

The IC supports heterogeneous subnets as long as all machines in a subnet are homogeneous. For example, certain subnets have 13 machines while others have 40, certain subnets have different disks and IO throughput and charging is based on the number of nodes in a subnet (i.e., replication factor). Overall horizontal scalability is achieved through the sharding mechanism. This effectively allows the IC to scale horizontally adding massive numbers of nodes without much additional overhead.

2.1 Failure Model

To maximize decentralization, the IC is designed for Byzantine fault-tolerance, in which faulty nodes may deviate in an arbitrary way from the IC protocol.

In any given subnet with $n \geq 3f+1$ nodes, at most f nodes may behave in a faulty manner. This is the highest number of failures which can be tolerated without additional assumptions on failures and message delivery [22, 48]. The failures account for software bugs, power outages as well as outright malicious behavior by colluding nodes. To limit the exposure and maximize decentralization, the nodes in a subnet are chosen in different geographical areas, jurisdictions and node provider organizations. In the future, trusted execution environments will further reduce the attack surface.

Traditional systems [8, 11, 38] often assume a weaker crash-stop failure model and aim to be available if a subset of nodes crash, but cannot cope with Byzantine behavior. The performance implications of Byzantine fault tolerance over crash-stop are acceptable for the applications deployed on the IC.

2.2 IC interface

The Internet Computer provides two distinct types of calls (i.e., requests sent to canisters): update and query calls. We refer to these operations interchangeably as either calls, requests, or messages. The IC also provides special calls for the canister life cycle: canister *creation*, canister *installation* and canister *upgrades*. Those are special forms of update calls.

Update calls can modify canister state. They are executed on all machines in a subnet participating in state machine replication. The calls are ordered and validated by consensus in a Byzantine fault-tolerant manner. This order, together with deterministic execution of canister code and relevant parts of the IC, provide state machine replication guarantees. Since consensus is computation and communication heavy, agreement and execution of update calls is done in batches to optimize throughput. Thus, update call latency is dependent on the time spent for consensus to reach agreement on blocks.

While update calls for different canisters may be executed in parallel, update calls for the same canister are processed sequentially. The response to an update call is threshold-signed by $2f+1$ nodes, i.e., a super-majority of the nodes created a signature collectively, hence users can verify correctness without having to communicate with multiple nodes.

The IC API guarantees atomicity for update calls as long as no further calls to other canisters are made. Updates that modify local state and run local computation are always atomic. For computation that calls into other canisters/smart contracts 2PC protocols could be implemented.

Query calls, on the other hand, do not change the canister's persisted state. As such, a query call may be processed directly by a single replica without passing through consensus. This reduces the query call latency significantly.

The correctness of query call responses from individual machines can be verified despite the Byzantine fault tolerance failure model with *certified variables*. Such variables carry threshold signatures which are generated collectively by a super-majority of the nodes in a subnet. Data elements that programmers want to verify via certified variables need to be arranged in a Merkle tree [35]. With certified variables, elements of a canister's state can be verified by clients even when talking to a single IC node.

Note that the Internet Computer does not guarantee any

order between query calls and other calls to the system (neither query nor update). If the order of calls matters, canister developers must use update calls and/or provide a versioning scheme as part of the canister code.

Applications running on different subnets can call each other by means of an asynchronous pull-based reliable communication primitive on top of consensus on both the sending as well as the receiving subnetwork.

IC Programming. Currently, developer support for applications programmed in *Rust*, *Motoko* [18], or Python [15] exists. The IC canister code is compiled down to WebAssembly, which is executed under a sandboxed virtual machine on the IC nodes. Any language that can be compiled down to WebAssembly could also be used. A detailed description of WebAssembly execution is provided in Section 3. Listing 1 shows an example of a functional 15-line key-value store canister implemented in Rust. It exports one update call to insert elements in the kv-store and one query call to retrieve them.

2.3 The IC Protocol Stack

As illustrated in Figure 1, the Internet Computer Protocol consists of four layers.

Peer-to-peer Layer. Within a subnet, nodes exchange information to achieve consensus on the replicated state and the messages to be processed next. To this end, the peer-to-peer layer offers a (prioritized) broadcast service to the layers above. To conserve bandwidth, peer-to-peer relies on an advert-based mechanism, where nodes first send a small advert to announce they have an artifact. Other nodes can then request the artifact if they need it, based on the details in the advert. Peer-to-peer relies on TLS over TCP streams between the nodes of a subnet. On top of that, it adds further reliability with notifications for unsent messages (in case of sender-side errors), and automatic connection re-establishment and requests for recent adverts.

Consensus Layer. Incoming messages must be validated and ordered so all replicas process them in the same order. The IC uses a novel consensus protocol [9] briefly described here.

The protocol proceeds in rounds. The replicas grow a tree of blocks referencing valid predecessor blocks. Their local trees form a consistent yet sometimes locally incomplete tree view. In each round, a pseudo-random process is used to assign each replica a unique rank. The replica of lowest rank is the *leader* of that round. When the leader is honest and the network is synchronous, the leader will propose a block, which the other honest nodes in the subnet will validate and add to their local tree. If the leader is not honest or the network is not synchronous, some other replicas of higher rank may also propose blocks, have them validated and added to the tree. Whenever $2f+1$ replicas report that they added exactly one block to the tree, this block and its predecessors on the path to the root are declared finalized and the non-finalized parts of the tree up to this height are pruned.

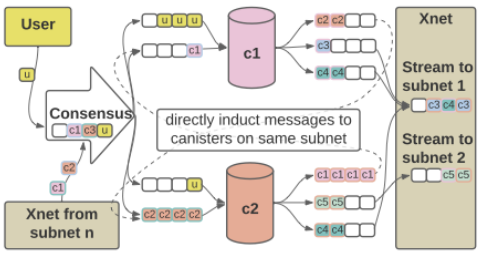


Figure 2: Routing messages through the IC protocol stack. Messages for canisters, issued by users or canisters on other subnets, are validated and ordered by consensus. Subsequently, messages are put into input queues for their destination canister. Messages created by canisters are put into output queues from where they are either transferred to their respective input queues on the same canister (bypassing consensus) or sent as part of streams to their target subnet.

One can prove that this protocol provides the consensus properties, namely *safety* (i.e., all replicas in fact agree on the same ordering of inputs) and *liveness* (i.e., all replicas should make steady progress). The IC consensus protocol guarantees safety despite asynchrony. This means that there is no assumption of an upper bound on the time to send information from one node to another. For liveness short intervals with fast message delivery are sufficient. The IC consensus protocol degrades gracefully when some replicas are malicious.

Message Routing. Once the consensus layer orders input messages, they are delivered to the message routing layer. The destination canister for each message is selected and the messages are enqueued for processing by the execution environment. During execution, the destination canister updates its state as part of the replicated state machine and generates outputs handed back to the message routing layer.

The message routing layer enqueues messages in one of multiple input queues. For each canister C running on a subnet, there are several input queues — there is one queue specifically for user-generated messages to C. Furthermore, each other canister C', from which C receives messages, gets its own queue. In each round, the execution layer will consume some of the inputs in these queues, update the replicated state of the relevant canisters, and place outputs in various output queues. For each canister C running on a subnet, there are several output queues — each other canister C', with whom C communicates, gets its own queue. The message routing layer will take the messages in these output queues and place them into subnet-to-subnet streams to be processed by a crossnet transfer protocol, whose job it is to actually transport these messages to other subnets. This is visualized in Figure 2.

Thus, the replicated state comprises the state of the canisters, as well as "system state", including the above-mentioned queues and streams. Thus, both the message routing and execution layers are involved in updating and maintaining the replicated state of a subnet. It is essential that all of this state

is updated in a completely deterministic fashion, so that all replicas maintain exactly the same state.

The consensus layer is decoupled from the message routing and execution layers, in the sense that only messages from finalized blocks of the chain reach routing and execution. Temporary block tree branches are pruned before their payloads are passed to message routing. This is in contrast to other blockchains which execute blocks speculatively, before ordering and validating them [46].

Execution Layer. The Execution Environment operates in rounds, during which it takes messages from canister input queues and executes the corresponding Wasm function with the message as payload. Based on the input and canister state, the execution environment updates the canister state, and could additionally add messages to output queues. One of the main challenges is that computation must be deterministic for state machine replication to work.

A scheduler determines in which order messages are executed in each round. The main goals of the scheduler are (see Section 3.3 for a more detailed description): (1) it must be *deterministic*; (2) it should distribute workloads *fairly* among canisters (3) optimizing for *throughput* over *latency*.

The IC offers orthogonal persistence, an illusion given to programs to run forever: the heap of each canister is automatically preserved and restored the next time it is called. Listing 1 shows an example key-value store that illustrates how easy it is to use orthogonal persistence. The key-value store in this case is backed by a simple Rust HashMap stored on the Wasm heap by means of a thread-local variable. We use a RefCell to provide interior mutability. The example would also be possible without it, but mutating the thread-local variable would be unsafe in that case, as the Rust compiler cannot guarantee exclusive access to it.

3 Systems Challenges of the IC

We focus on the execution layer of the IC and discuss the challenges C1-C4, as well as their solutions.

The IC can execute arbitrary programs. The basic computational unit in the IC is called a canister. Canister programs are encoded in WebAssembly (Wasm) [28], a binary instruction format for a stack-based virtual machine.

The main goal is to execute `deterministically`, `securely` and `efficiently` the functions triggered by messages sent to canisters. Each canister is executing under a long-running Wasm virtual machine whose state is persisted over long periods of time. In terms of efficiency IC nodes are able to sustain running tens of thousands of canisters [17]. The *memory subsystem* of the nodes addresses challenge **C1**.

The IC needs to scale up, by achieving high resource utilization on individual nodes. This is important to achieve performance comparable to native systems and to amortize the cost of state machine replication. Essential for achieving

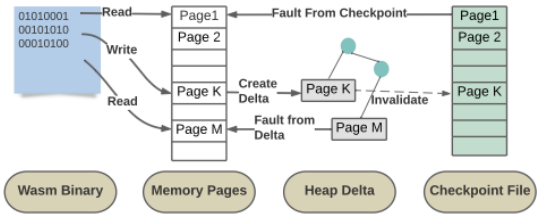


Figure 3: Memory faulting architecture, including heap delta and checkpoint file. When Wasm instructions trigger page faults, memory contents can be faulted in from the memory checkpoint. When pages are dirtied by writes, heap deltas are created, which invalidate page content in the checkpoint file. Subsequent faults are served directly from heap deltas. Periodically, heap deltas are merged into a new checkpoint.

these goals is to enable efficient execution of developer code through Wasm code execution, solving challenge **C2**.

To ensure correct and deterministic state machine replication, we designed and implemented a deterministic scheduler for the IC nodes. Our scheduler implements a deterministic time slicing mechanism, effectively solving challenge **C3**.

Security is achieved at multiple layers of the IC through trust, consensus, byzantine fault-tolerance and so forth. Details about these can be found in our whitepaper [53]. At this layer, we ensure security through operating systems and virtualization mechanisms effectively solving challenge **C4**.

3.1 C1 - Statefulness - The Memory Subsystem

Currently, canisters can use up to 52 GiB of memory to be accessed by users. Any implementation of orthogonal persistence has to solve two problems: **(1)** How to map the persisted memory into the Wasm memory; and **(2)** How to keep track of all modifications in the Wasm memory so that they can be persisted later. We use page protection to solve both problems. We divide the entire address space of the Wasm memory into 4 KiB pages. All pages are initially marked as inaccessible using the page protection flags of the OS.

The first memory access triggers a page fault, pauses the execution, and invokes a signal handler. The signal handler then fetches the corresponding page from persisted memory and marks the page as read-only. Subsequent read accesses to that page will succeed without any help from the signal handler. The first write access will trigger another page fault, however, and allow the signal handler to remember the page as modified and mark the page as readable and writable. All subsequent accesses to that page (both r/w) will succeed without invoking the signal handler.

Invoking a signal handler and changing page protection flags are expensive operations. Messages that read or write large chunks of memory cause a storm of such operations, degrading performance of the whole system. This can cause severe slowdowns under heavy load.

Versioning: Heap Delta and Checkpoint Files. A canister

Figure 4: The performance improvement given by memory faulting optimizations (lower is better). Note the logarithmic vertical axis. Speedups range from 1.25X to 3.5X.

executes update messages sequentially, one by one. Queries, in contrast, can run concurrently to each other and to update messages. The support for concurrent execution makes the memory implementation much more challenging. Consider that a canister is executing an update message at (blockchain) block height H. At the same time, there could still be a previous long-running query that started earlier, at block height $H-K$. This means the same canister can have multiple versions of its memory active at the same time; this is used for the parallel execution of queries and update calls.

A naive solution to this problem would be to copy the entire memory after each update message. That would be slow and use too much storage. Thus, our implementation takes a different route. It keeps track of the modified memory pages in a persistent tree data-structure [41] called Heap Delta that is based on Fast Mergeable Integer Maps [37]. At a regular interval (i.e., every N rounds), there is a checkpoint event that commits the modified pages into the checkpoint file after cloning the file to preserve its previous version. Figure 3 shows how the Wasm memory is constructed from Heap Delta and the checkpoint file.

Memory Faulting Optimizations. We describe below three optimizations we designed to improve memory faulting.

◊ **Optimization 1:** *Memory mapping the checkpoint file pages.* This reduces the memory usage by sharing the pages between multiple calls being executed concurrently. This optimization also improves performance by avoiding page copying on read accesses. The number of signal handler invocations remains the same as before, so the issue of signal storms is still open after this optimization.

◊ **Optimization 2:** *Page tracking in Queries.* All pages dirtied by a query are discarded after execution. This means that the signal handler does not have to keep track of modified pages for query calls. As opposed to update calls, for queries we introduced a fast path that marks pages as readable and writable on the first access. This low-hanging fruit optimization made queries 1.5x-2x faster on average.

◊ **Optimization 3:** *Amortized prefetching of pages.* The idea behind the most impactful optimization is simple: to reduce the number of page faults, we need to do more work per signal handler invocation. Instead of fetching a single page at a time, the signal handler tries to speculatively prefetch pages. The right balance is required here because prefetching too many pages may degrade performance of small messages that access only a few pages. The optimization computes the largest contiguous range of accessed pages immediately preceding the current page. It uses the size of the range as a hint for prefetching more pages. This way the cost of prefetching is amortized by previously accessed pages. As a result, the optimization reduces the number of page faults in memory intensive messages by an order of magnitude.

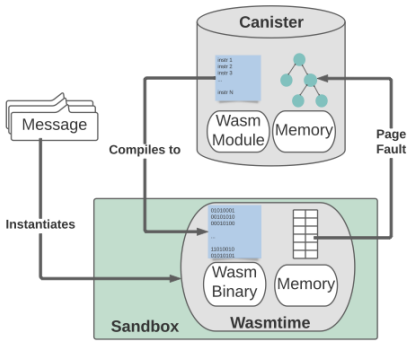


Figure 5: The execution of messages instantiates a Wasmtime instance in a sandboxed environment. Each sandbox can run multiple wasmtime instances. The Wasm module is compiled to a binary running inside the VM while memory accesses are faulted in from the memory heap deltas and checkpoint file.

These optimizations bring substantial benefits for the performance of the memory faulting component of the execution environment. Figure 4 plots the performance optimizations we achieved when enabling all three optimizations in comparison with turning them off. We measured this for a memory intensive benchmark which allocates 8 bytes, 1 MiB, or 512 MiB. The optimizations allow the IC to improve its throughput for memory-intensive workloads as depicted in the Figure and no performance degradation was observed for other workloads.

3.2 C2 - Scalability: Wasm Execution

To process a message, the canister executes the corresponding function in the Wasm module. Figure 5 depicts this process. Function execution requires a Wasm instance which is a combination of Wasm code and memory. One of the challenges is that we cannot afford to keep one Wasm instance alive for each of the canisters running in a subnet because we would run out of memory. Instead, we construct Wasm instances on demand for each message and dispose of them after the message execution. Thus, the latency of message execution depends on the instantiation time and the actual time to execute the function. Included in this instantiation time is also the time to compile the Wasm code. One optimization we deployed is to cache compilations of Wasm code.

For one of the most used canisters in production, the compilation cache optimization reduces the P99 for running non-replicated queries by 2 orders of magnitude. This is depicted in Figure 6. Therefore, for our real-world example, cold-start times are now below 10 ms for previously compiled user-code

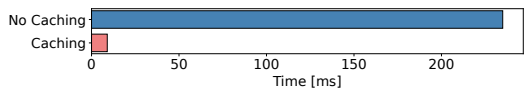


Figure 6: Compilation caching effects. P99 for running a short (1 ms), non-replicated query on a cold started canister.

and around 230 ms when compiled for the first time. This is an important achievement compared to major serverless providers where cold-start times are in the order of seconds to tens of seconds [1, 49, 51, 55]. However, compilation time varies proportionally with the complexity of the Wasm code being compiled. Therefore, for certain applications longer first-time compilation times are to be expected—subsequent calls are optimized by caching the compilations.

Section 3.1 describes how the Wasm instance manages a canister's memory in checkpoint files or heap deltas.

Instrumentation: Time-slicing and Accounting. Since Wasm is Turing-complete we need a mechanism to ensure termination of message execution. Otherwise, a faulty or malicious smart contract would be able to stall the progress of the blockchain by potentially infinitely long running messages. As everything else in the execution layer, the point at which we do that needs to be deterministic. However, execution duration is not, as execution might be slightly different on different nodes due to nondeterministic events in the system (e.g. one machine having a page fault, but not the other one).

Instead, we instrument Wasm code to count the number of instructions that have been executed for each message that is being processed by the system. To reduce the performance overhead, our compiler extension performs counting at the basic block level instead of individual instructions. Concretely, at the start of an execution round we initialize the global instruction counter of the Wasm module for each canister to the instruction limit. In each basic block of the Wasm module we insert a snippet of code to decrement the counter by the number of instructions in the basic block. In re-entrant blocks, such as function and loop headers, we insert code that aborts message execution when the counter is negative.

Another benefit of quantifying computation done via instruction counting is that we can deterministically charge canisters for performed work. The canisters are charged for the resources they are consuming, including computation, communication and storage. For that reason, the IC needs to account for resource usage of all canisters in the system. Two examples for resources being accounted are memory accesses (estimated by the number of pages read and written) as well as CPU instructions used. Memory accesses are tracked with the memory protection mechanisms as described in Section 3.1.

Resources also need to be accounted for when serving users query calls. Queries are especially complex since their execution is non-replicated due to their execution on just a single node. However, the canister still needs to be charged deterministically by all nodes as the balance of a canister is part of the canister's state which is managed by state machine replication. This is currently an open problem we are investigating.

3.3 C3 - Deterministic Scheduling

Since we are operating a replicated state machine, it is essential that each replica processes the same inputs in the same order. To achieve this, the replicas in a subnet run a consensus protocol [21], which ensures that they process inputs in the same order. If the IC code executing those messages as well as the canister code itself is deterministic, the internal state of each replica will evolve over time in exactly the same way, and each replica will produce exactly the same sequence of outputs in the absence of hardware-related problems.

Granularity. For simplicity, the scheduler works at a coarse level, scheduling canisters instead of individual messages and executing each canister until there are no more messages in its queues or the system-defined instruction limit for a round is reached. IC nodes are modern dual-socket multi-core servers. Deterministic behavior on such a machine can be achieved when canisters are pre-allocated to specific CPU cores at the beginning of each round. Our current scheduler takes this approach because it is a simple and effective design choice which is then easily proven correct (see Appendix A).

Allocation and fairness. To ensure responsiveness under heavy load, canisters have the option of paying upfront for a *compute allocation*. Since canisters are single threaded, a *compute allocation* is a fraction of one CPU core, expressed in percentage points. Only part of a subnet's compute capacity can be allocated, ensuring progress for canisters with zero compute allocation, i.e. best effort canisters. Fairness is defined as guaranteeing canister compute allocations (i.e., a backlogged canister with compute allocation A executing at least A full rounds out of every 100) and evenly distributing the remaining capacity ("free compute") across all canisters.

Given a deterministic state machine with N CPU cores (and $N \times 100$ compute capacity), we schedule (at least) N canisters to execute a *full round*: a round, in which a canister either exhaust the instruction limit or completes the execution of all their enqueued messages. The scheduling algorithm uses credits accumulated across rounds as priority: an amount of credits equal to the canister's compute allocation plus a uniform share of the *free compute* is credited to every canister at the beginning of every round; canisters in the priority queue are assigned round-robin to CPU cores (each of the first N canisters are scheduled first on a CPU core), and 100 credits are debited from each canister that executes a full round.

Our algorithm's time complexity to compute the schedule algorithm is linear in the number of canisters, which is acceptable because scheduling only happens once per round (alongside other operations that require linear time). The scheduling algorithm has the following properties:

- `Correctness wrt. compute allocation`: every backlogged canister gets a "full execution round" at least A out of every 100 rounds, where A is the canister compute allocation.

- `High throughput`: in the absence of information regarding the number of instructions required to execute each message ahead of time (which might allow for better bin packing); and given the constraint that canisters must be allocated to specific cores ahead of time; the algorithm provides optimal throughput by executing canisters allocated to each core until the round instruction limit is reached.
- `Fairness`: the credits system ensures that (backlogged) canisters with equal compute allocations get the same number of "full execution rounds" over a long enough time period.

Appendix A defines and anlyses the scheduler formally.

Deterministic Time Slicing. The scheduler, as explained above, although effective, is suboptimal for messages that trigger executions of varying length. Each subnet of the IC operates in epochs of many rounds. Messages run either to completion or to a predefined upper limit on the number of instructions per round. In the second case, the execution is aborted and returns an error. In this case, the user would have to rewrite the algorithm to make the execution of the long-running message span multiple execution rounds. This is less than ideal because of two reasons. First, it can artificially increase round duration due to stragglers, which leads to overall throughput loss and slowdown. Second, it can lead to significant CPU waste because a long-running execution that is aborted due to reaching instruction limits will be re-executed.

To solve this problem, we designed a *deterministic time slicing* mechanism on top of our scheduler, where each message execution longer than a round is `sliced` in a number of intervals with roughly equal numbers of instructions. This is akin to time slicing in modern schedulers [7], although the biggest challenge here is to enforce determinism. Time slicing also increases the amount of intermediate state, that needs to be kept while a message is preempted. The slices achieved here are then scheduled as described before.

3.4 C4 - Ensuring Security

The security model of the IC aims to provide *access control* and *integrity* of canister and system data in the presence of malicious canisters and users. Canisters can specify a method that accepts or rejects requests, e.g., based on caller ID, resource consumption etc. Only correctly signed messages are then processed. Responses to update calls and queries for certified variables are threshold-signed by the subnet nodes, so clients can verify authenticity. Canisters cannot inspect or change state of other canisters or parts of the system. This is guaranteed by eliminating the main Wasm attack vectors.

An adversary may craft Wasm code to: (1) exploit bugs in the Wasm engine to escape its protection mechanism; and (2) perform side-channel attacks to obtain data from the system or other canisters [33]. The IC protects against these attacks using OS isolation and sandboxing. Each canister is compiled and executed in its own sandboxed process that communicates only with the main replica process via security-audited IPC.

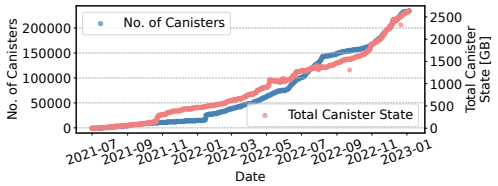


Figure 7: Total number of canisters running on the IC.

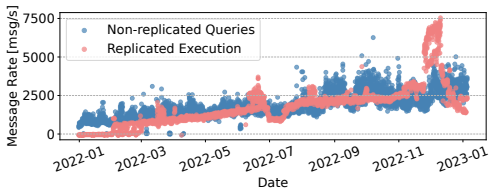


Figure 8: The rate of non-replicated and replicated messages.

Sandboxes are given minimal permissions needed to execute using object-based access control (SELinux).

In the future, hardware-based security, offering fully encrypted VMs with the possibility to attest remotely if the expected VMs are running, will increase obstacles curious and malicious node providers face.

4 The Internet Computer In Data

We present data related to the operation and performance of the IC. We show the growth and usage patterns the IC is experiencing, its overall performance (in comparison with native code) and identify sources of overhead with regard to the systems challenges presented in Section 3.

The Internet Computer Hardware. The IC currently runs on homogeneous hardware that is hosted by independent node providers. The chosen configuration makes use of dual-socket AMD EPYC 7302 processors with a total of 32 physical cores running at 3 GHz, each core having 2 hardware threads. The IC servers make use of 503 GiB of memory. AMD chips were chosen due to their secure encrypted virtualization feature to enable VM encryption across VM upgrades in combination with remote attestation.

The data presented in Sections 4.1-4.2 is gathered from production. Experiments discussed in the rest of this section are executed on an internal testnet that mimics subnets on the IC. The difference to IC machines is that two IC VMs are deployed to each host, instead of one. Testnet machines are hence expected to be slower for concurrent workloads.

4.1 A High-level View of the IC

IC Growth (C1 + C2). We focus on the high-level operational data gathered from the IC. The usage has been steadily increasing since launch, showing an acceleration of the numbers of deployed applications since the beginning of 2022. Figure 7 depicts the number of deployed canisters over time, as well as their overall allocated state, which reaches 2.5TB.

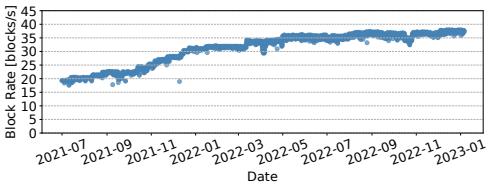


Figure 9: The block rate of the IC.

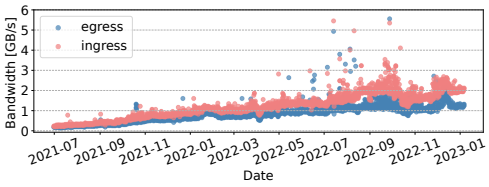
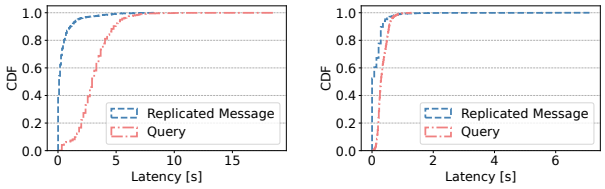


Figure 10: Aggregated IC Nodes network traffic.

Workload Growth (C1 + C2). Together with the increase in the deployed applications we also observe an increase in the workload deployed on the IC. Figure 8 plots the arrival rate of messages over time. As expected, non-replicated messages (i.e., queries), which do not pass through consensus and do not alter canister state are being triggered significantly more often than replicated messages by our users. Replicated execution, which incurs the consensus overhead is used less — we assume only for operations that need to alter application state. An interesting observation to make here is that in February 2022, we have changed the way in which we quantify the number of replicated messages. Whilst before this date, the number only sums up *update* calls, after this date the date adds also replicated execution that the canisters use for their operation (e.g., periodic heartbeats). The graph shows a significant increase in the number of replicated messages after February 2022. We note here that it is common practice that for *operational systems*, metrics sometimes change meaning over time as the systems itself is refined and continuously evolving. The significant increase in replicated execution in Dec 2022 is due to the launch of a popular application, while the later drop corresponds to a change in the call pattern of the same application.

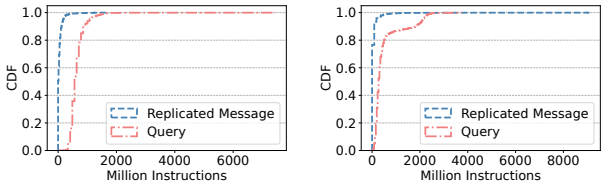
IC Scaling Out (C2). Over time, the IC has increased its capacity to sustain increased workloads and achieve decentralization. Evidence to sustain the scaling out of the IC is given by examining the *block rate* – the number of blocks generated by the consensus layer. Figure 9 plots these data, showing an increase of 57% since launch.

Similar evidence to sustain growth of the IC in terms of higher workload demands is represented by the increased network bandwidth used to exchange messages between nodes. Figure 10 depicts the aggregated bandwidth of traffic generated: since launch, the traffic generated between the nodes has increased from around 250 MB/s to about 3 GB/s, over an order of magnitude increase over a period of a year. The amount of ingress and egress traffic is very similar, as expected.



(a) OpenChat Subnet. *(b) DSCVR Subnet.*

Figure 11: The time queries or replicated messages take in the execution layer, without consensus overhead.



(a) OpenChat Subnet. *(b) DSCVR Subnet.*

Figure 12: The number of instructions spent for executing queries or replicated messages.

4.2 The IC Performance

The IC has attracted the developers of several large applications, for example: *OpenChat* (a decentralized chat application), *DSCVR* (decentralized social news aggregator), and *distrikt* (a decentralized professional social media platform). These applications have added a significant increase in workload complexity for the IC. We present an in-depth analysis of the metrics of the subnet that runs the OpenChat canisters. We focus mostly on data related to the systems challenges described in Section 3.

The subnet hosting the OpenChat application is composed of 13 replica nodes distributed geographically (in North America, Europe and East Asia). The subnet hosts 80,000 canisters.

Replicated vs. Non-Replicated Execution. (C1-C3) First, we assess the duration of non-replicated queries in comparison with replicated update calls in Figure 11. We compare the data in the OpenChat subnet with the data coming from the subnet running DSCVR (and other applications). Note that this comparison does not include the overhead of the P2P, consensus and messaging layer, but only measures the time spent in the Execution layer. By analyzing the data in Figure 11 we conclude that queries run longer for the OpenChat subnet, while updates dominate on the DSCVR subnet. Figure 12 plots the number of Wasm instructions for queries compared to replicated messages. The average number of instructions executed for queries is significantly higher than for replicated messages for OpenChat, leading to longer executions. On the contrary, on the DSCVR subnet the behavior is the exact opposite. This shows that the IC runs a diverse set of applications, with varied needs and characteristics.

Memory Overhead (C1 + C2). The heap delta is used in-between checkpoints to keep track of modified memory pages. This data structure can affect the subnets' ability to scale up

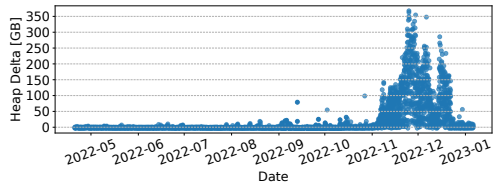


Figure 13: Heap delta for an IC subnet over time.

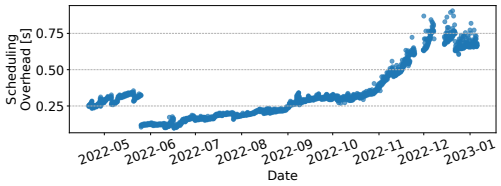


Figure 14: Scheduling overhead for an IC subnet over time.

to many concurrent canisters as tracking modifications incurs overhead. Figure 13 shows how the heap delta for all canisters on OpenChat subnet evolves over time. The data shows that on average the heap delta is below 5 GB, but more recently, since November 2022 it has significantly increased, up to 350 GB. This shows a large increase in OpenChat usage, aggregated for more than 80,000 canisters. On subnets with fewer canisters heap deltas are typically smaller.

Scheduling Overhead (C3). When scheduling replicated messages the same ordering has to be ensured on all replicas so that determinism is guaranteed. Deciding which message gets scheduled for execution and at what time is a costly operation that can affect scalability. We plot the scheduling overhead, i.e., the time to compute the schedule for one round, in the OpenChat subnet in Figure 14. The immediate conclusion is that this overhead is in the order of hundreds of milliseconds per round for this subnet. An interesting point is the overhead reduction which can be observed around the end of May 2022. This is attributed to an optimization related to the scheduling process, namely checking if a canister has messages to be run. We checked the scheduling overhead for subnets with smaller numbers of canisters as well. Our conclusion is that this overhead is proportionally smaller (as the overhead complexity is $O(N \log N)$ for N canisters). Similarly to the heap delta observation, the scheduling overhead increased significantly since November 2022. This is also because the number of users (and canisters) for OpenChat has significantly increased, leading to many more messages being scheduled for execution in this subnet. We investigate ways to reduce this overhead to support efficient scheduling with larger workloads.

4.3 The IC Virtualization Stack

We quantify the impact of the IC virtualization. In Section 3 we described how user code is executed. In short, user code is compiled to Wasm, which gets instrumented and compiled to a binary that gets executed inside a sandbox. To achieve orthogonal persistence and stateful execution we keep track of memory writes in a persistent data structure from which the Wasm VM is faulting in its pages. This indirection introduces a non-trivial overhead. We quantify this overhead for two types of workloads: compute- and memory-intensive. Each of these workloads stress different resources of the stack.

Compute Intensive Workload (C2). We implemented a workload that calculates prime numbers up to a given integer number. This is a single threaded workload, which we wrote in Rust and deployed on the IC. We ran the same workload (identical Rust code) on an IC machine natively (compiled to an x86 binary), without the entire virtualization stack. Furthermore, we ran the same Rust code on one of the top-3 serverless providers. Experiments in the IC have been measured from within the Execution Environment and hence do not contain network latency or the cost of other parts of the IC stack. We provide similar data for the serverless provider and take the latency from the provider's dashboard. The experimental data is presented in Table 1. The overhead is computed against the native execution (not against the serverless provider).

First, the Internet Computer performance compared to native execution is good for longer-running workloads considering the extra features that the IC execution environment provides: sandboxing, accounting and tracking changes. Second, we emphasize that the IC performance is in the same order of magnitude with one of the top-3 serverless providers. Considering the extra operations that the IC does to offer its users decentralization, security etc., we deem these performance data encouraging, especially taking into account the fact that the serverless execution is faster than native execution in our case. This directly implies that the hardware running the serverless platform is very likely faster than the hardware we described in Section 4.

Memory Intensive Workload (C1). We performed a similar experiment with a memory intensive workload. Here, memory is accessed sequentially in strides of 8 bytes. The totally allocated memory is 1 GB. In this experiment we only compare against a native execution because all current serverless platforms are not stateful. Therefore, the serverless functions access memory directly (i.e., without any faulting architecture, persistence, versioning) through either microVMs [1] or containers [43]. A more direct comparison would involve a serverless function that stores its state in a storage environ-

n	**IC** [ms]	**Native** [ms]	**Serverless** [ms]	**Slowdown IC / native**
0	1.40	0.02	3.53	70 X
100	1.43	0.03	1.93	47 X
1,000	2.35	0.94	2.94	2.5 X
10,000	41.54	33.85	19.65	1.22 X
50,000	718.26	610.73	347.54	1.17 X

Table 1: Median computation time for a compute intensive workload running on the IC, native execution and running on a serverless provider (average for serverless due to lack of raw data) over 30 executions. The workload identifies primes in the first n integers.

Operation	Data Size [Bytes]	IC [ms]	Native [ms]	Slowdown IC / native
Read	50,000	2.15	0.02	107 X
Read	5,000,000	26.36	1.83	14.2 X
Read	50,000,000	195.52	18.27	10.7 X
Write	50,000	2.28	0.02	114 X
Write	5,000,000	33.79	2.14	15.8 X
Write	50,000,000	277.36	19.13	14.5 X

Table 2: Median computation time over 30 executions of a memory intensive benchmark performing strided reads/writes of different sizes on the IC and native execution.

ment [32], but this is outside the scope of this article.

We therefore compare the execution time of executing the benchmark compiled to a native x86 binary on one of the IC node to the time of executing the same benchmark as canister code. Table 2 summarizes our findings. For workloads that touch up to 50 MiB of data, the overhead of running this benchmark on the IC is approximately 10 X to 15 X. Lower amounts of data touched incur larger slowdowns, with smaller data giving the largest slowdown.

Even though the slowdown compared to native execution seems large, we remind the reader that a rather deep virtualization stack is involved in memory operations. Further, the IC needs to account for resource consumption and track memory writes, which the native version does not do. Finally, update calls pass through consensus and the entire IC stack, therefore offering the users all the Internet Computer benefits: decentralization, security, and tamper-proof execution. Moreover, we remind the reader that the IC is orders of magnitude faster and more efficient than other blockchains. We are further working on improving the memory faulting layer using write barriers [6] or userfaultfd [39] so that in the future we can reach our goal of (close-to-)native performance.

The Cost of Decentralization and Statefulness (C1-C4). One of the overarching goals of the IC is to offer levels of performance as close as possible to *native* and traditional client-server architecture performance. With regard to user-perceived overhead, the factors contributing most are the consensus protocol, the networking, and the crypto primitives, as well as memory faulting. At a macro level, this overhead can be observed by re-interpreting Figures 9 and 10. All the replication protocol-related mechanisms leads to network traffic (e.g., a few MB per machine per second, see Figure 10, considering that at the moment of writing the IC runs on over 500 machines) and computational overhead for the creation and validation of blocks and the messages contained therein. This overhead is not crippling the operation of the IC and its benefits significantly outweigh its downsides.

We ran many memory-intensive *update* calls in one of our testing and benchmarking subnets. Memory-intensive updates especially stress the entire system stack because they: (i) modify significant amounts of the canister state; (ii) need replicated execution; (iii) require consensus for ordering. Therefore, we quantify the overhead of system components by running `Linux perf`. Figure 15 is an instance of quantifying

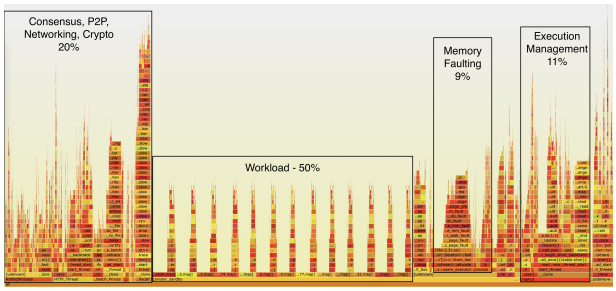


Figure 15: Decentralization and statefulness overheads when several memory-intensive `update calls` are made. Data presented as flame graphs [26].

such overhead. We observe that the actual workload takes approximately 50% of the used CPU time (not all the CPU capacity is used). A large fraction of the overhead can be attributed to the consensus and P2P protocol, networking or crypto primitives stack, together they account for roughly 20% of the CPU time. Another sizeable CPU time consumption is due to the memory faulting subsystem (9%) and the execution management stack (11%). The latter involves all processing related to canister administration, Wasm instrumentation, communication with sandboxes.

4.4 End-to-end Performance (C1-C4)

We quantify the end-to-end performance of our subnets using the default subnet size of 13 nodes. Note that for enhanced security guarantees (e.g., tolerating more malicious nodes) the IC hosts even larger subnets (i.e., 40 machines). All subnets are geo-replicated, i.e., are composed of machines running on multiple continents – the Americas, Europe, Asia. Our results are depicted in Table 3.

In contrast to other experiments we execute requests with insignificant execution overhead, so we can safely attribute the latency overhead to other layers. A geo-replicated subnet is able to run ∼78K queries per second with a latency of 50ms-200ms, given by differences in geographic location of client and targeted IC node. Since no coordination among nodes is needed for query execution, we can safely attribute this latency to the networking layers. In terms of updates (stateful and replicated execution), a geo-replicated subnet is able to serve 950 updates per second for a latency of 1-4s (which

Op	Throughput (ops / s)	Latency (s)	Overheads
Query	78,000	0.05-0.2	Networking
Update	950	1-4	Networking, Consensus, Replicated Execution, Statefulness

Table 3: End-to-End performance for the two operations supported by the IC.

includes networking, consensus and replication overheads). Note that this latency is comparable to the top-3 serverless platforms cold starts [54].

5 Related Work

The IC builds on fruitful years of research at many levels: from consensus, to peer-to-peer networking, cryptographic protocols, blockchain, and operating systems. We limit our discussion to blockchain-related technology and large-scale systems making use of it, and the value of opening up and discussing data from large-scale operational systems.

Blockchain Execution Environments. It has become a mantra that blockchains are slow. Just like for the IC, others have investigated ways in which computations and transactions running atop blockchains can be sped up. These are related to either speeding up consensus [12, 52], using software transactional memory [25,46,47], enforcing determinism and ordering [34, 52], or optimizing execution layers [2]. The performance evaluation in these works relies on synthetic workloads in test environments. To the best of our knowledge, this paper is the first to report on the performance of the execution environment of a real-world blockchain deployment.

Operational Systems and Data. Next to the more traditional workloads, such as high-performance scientific computing [20], analytics [4], cluster workloads [45], recently data centers started providing Blockchain-as-a-service offerings [23]. As Amvrosiadis et al. point out [3] data set diversity is key to understand the characteristics of workloads and to tailor new resource management schemes. The community recognizes and emphasizes the need of analyzing operational systems, such as the Microsoft Serverless platform [49], CloudLab [19], or the evolution of the Google data center network [40]. We believe our article adds significant data and insight on the design, operation and growth of systems that offer general-purpose computation capabilities on top of blockchain platforms.

6 Conclusion

The Internet Computer overcomes traditional blockchain limitations with respect to speed, storage costs, and computational capacity. We demonstrated how the novel design of the IC, coupled with solving low-level systems challenges enables decentralized stateful serverless. In particular, we presented an in-depth description of the execution layer of the IC followed by an evaluation of operational and performance data over real-world workloads as well as compute and memory-intensive benchmarks.

The IC code and data can be found here:
- IC code: https://github.com/dfinity/ic
- Dashboard: https://dashboard.internetcomputer.org/
- Dataset API: https://ic-api.internetcomputer.org/api

References

[1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)* (2020), pp. 419–434.

[2] Amiri, M. J., Agrawal, D., and El Abbadi, A. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), IEEE, pp. 1337–1347.

[3] Amvrosiadis, G., Park, J. W., Ganger, G. R., Gibson, G. A., Baseman, E., and DeBardeleben, N. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 533–546.

[4] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (2015), pp. 1383–1394.

[5] Bano, S., Sonnino, A., Al-Bassam, M., Azouvi, S., McCorry, P., Meiklejohn, S., and Danezis, G. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies* (2019), pp. 183–198.

[6] Blackburn, S. M., and Hosking, A. L. Barriers: Friend or foe? In *Proceedings of the 4th international symposium on Memory management* (2004), pp. 143–151.

[7] Bouron, J., Chevalley, S., Lepers, B., Zwaenepoel, W., Gouicem, R., Lawall, J., Muller, G., and Sopena, J. The battle of the schedulers:{FreeBSD}{ULE} vs. linux {CFS}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 85–96.

[8] Brooker, M., Chen, T., and Ping, F. Millions of tiny databases. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (USA, 2020), NSDI'20, USENIX Association, p. 463–478.

[9] Camenisch, J., Drijvers, M., Hanke, T., Pignolet, Y.-A., Shoup, V., and Williams, D. Internet computer consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing* (2022), pp. 81–91.

[10] Carbon Crowd. CO2 emissions assessment of the internet computer. https://wiki.internetcomputer.org/wiki/L1_comparison, 2022.

[11] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst. 31*, 3 (aug 2013).

[12] Danezis, G., Kokoris-Kogias, L., Sonnino, A., and Spiegelman, A. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 34–50.

[13] Dang, H., Dinh, T. T. A., Loghin, D., Chang, E.-C., Lin, Q., and Ooi, B. C. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data* (2019), pp. 123–140.

[14] DEARLE, A., KIRBY, G. N., AND MORRISON, R. Orthogonal persistence revisited. In *International Conference on Object Databases* (2009), Springer, pp. 1–22.

[15] DEMERGENT LABS. Python cdk for the internet computer. https://github.com/demergent-labs/kybra, 2023.

[16] DFINITY FOUNDATION. Comparison between the internet computer and other l1 blockchains. https://wiki.internetcomputer.org/wiki/L1_comparison, 2022.

[17] DFINITY FOUNDATION. The internet computer subnets. https://dashboard.internetcomputer.org/subnets, 2023.

[18] DFINITY FOUNDATION. The motoko programming language. https://internetcomputer.org/docs/current/developer-docs/build/cdks/motoko-dfinity/motoko/, 2023.

[19] DUPLYAKIN, D., RICCI, R., MARICQ, A., WONG, G., DUERIG, J., EIDE, E., STOLLER, L., HIBLER, M., JOHNSON, D., WEBB, K., ET AL. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)* (2019), pp. 1–14.

[20] FEITELSON, D. G., TSAFRIR, D., AND KRAKOV, D. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing 74*, 10 (2014), 2967–2982.

[21] FISCHER, M. J. The consensus problem in unreliable distributed systems (A brief survey). In *Fundamentals of Computation Theory, Proceedings of the 1983 International FCT-Conference, Borgholm, Sweden, August 21-27, 1983* (1983), vol. 158 of *Lecture Notes in Computer Science*, Springer, pp. 127–140.

[22] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (1985), 374–382.

[23] GAI, K., GUO, J., ZHU, L., AND YU, S. Blockchain meets cloud computing: A survey. *IEEE Communications Surveys & Tutorials 22*, 3 (2020), 2009–2030.

[24] GARAY, J., AND KIAYIAS, A. Sok: A consensus taxonomy in the blockchain era. In *Cryptographers' track at the RSA conference* (2020), Springer, pp. 284–318.

[25] GELASHVILI, R., SPIEGELMAN, A., XIANG, Z., DANEZIS, G., LI, Z., XIA, Y., ZHOU, R., AND MALKHI, D. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. *arXiv preprint arXiv:2203.06871* (2022).

[26] GREGG, B. The flame graph. *Communications of the ACM 59*, 6 (2016), 48–57.

[27] GROTH, J., AND SHOUP, V. On the security of ecdsa with additive key derivation and presignatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2022), Springer, pp. 365–396.

[28] HAAS, A., ROSSBERG, A., SCHUFF, D. L., TITZER, B. L., HOLMAN, M., GOHMAN, D., WAGNER, L., ZAKAI, A., AND BASTIEN, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), pp. 185–200.

[29] HUANG, H., KONG, W., ZHOU, S., ZHENG, Z., AND GUO, S. A survey of state-of-the-art on blockchains: Theories, modelings, and tools. *ACM Computing Surveys (CSUR) 54*, 2 (2021), 1–42.

[30] JIA, Z., AND WITCHEL, E. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 691–707.

[31] JORDAN, M. J., AND ATKINSON, M. P. Orthogonal persistence for java—a mid-term report. *Morrison et al.[161]* (1999), 335–352.

[32] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 427–444.

[33] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., ET AL. Spectre attacks: Exploiting speculative execution. *Communications of the ACM 63*, 7 (2020), 93–101.

[34] MEHRARA, M., HAO, J., HSU, P.-C., AND MAHLKE, S. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM Sigplan Notices 44*, 6 (2009), 166–176.

[35] MERKLE, R. C. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology* (Berlin, Heidelberg, 1987), CRYPTO '87, Springer-Verlag, p. 369–378.

[36] NGUYEN, D., LENHARTH, A., AND PINGALI, K. Deterministic galois: On-demand, portable and parameterless. *ACM SIGPLAN Notices 49*, 4 (2014), 499–512.

[37] OKASAKI, C., AND GILL, A. Fast mergeable integer maps. In *Workshop on ML* (1998), pp. 77–86.

[38] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[39] PENG, I., MCFADDEN, M., GREEN, E., IWABUCHI, K., WU, K., LI, D., PEARCE, R., AND GOKHALE, M. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)* (2019), IEEE, pp. 71–78.

[40] POUTIEVSKI, L., MASHAYEKHI, O., ONG, J., SINGH, A., TARIQ, M., WANG, R., ZHANG, J., BEAUREGARD, V., CONNER, P., GRIBBLE, S., ET AL. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 66–85.

[41] PROKOPEC, A., BRONSON, N. G., BAGWELL, P., AND ODERSKY, M. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (2012), pp. 151–160.

[42] PSARAS, Y., AND DIAS, D. The interplanetary file system and the filecoin network. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)* (2020), IEEE, pp. 80–80.

[43] RANDAZZO, A., AND TINNIRELLO, I. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (2019), IEEE, pp. 209–214.

[44] RAVICHANDRAN, K., GAVRILOVSKA, A., AND PANDE, S. Destm: harnessing determinism in stms for application development. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), pp. 213–224.

[45] REISS, C., WILKES, J., AND HELLERSTEIN, J. L. Google cluster-usage traces: format+ schema. *Google Inc., White Paper 1* (2011).

[46] RUAN, P., LOGHIN, D., TA, Q.-T., ZHANG, M., CHEN, G., AND OOI, B. C. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 543–557.

[47] SAAD, M. M., KISHI, M. J., JING, S., HANS, S., AND PALMIERI, R. Processing transactions in a predefined order. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (2019), pp. 120–132.

[48] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (1990), 299–319.

[49] SHAHRAD, M., FONSECA, R., GOIRI, Í., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 205–218.

[50] SHILLAKER, S., AND PIETZUCH, P. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 419–433.

[51] SILVA, P., FIREMAN, D., AND PEREIRA, T. E. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 1–13.

[52] SURI-PAYER, F., BURKE, M., WANG, Z., ZHANG, Y., ALVISI, L., AND CROOKS, N. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 1–17.

[53] THE DFINITY TEAM. The Internet Computer for Geeks. Cryptology ePrint Archive, Paper 2022/087. https://eprint.iacr.org/2022/087.

[54] USTIUGOV, D., AMARIUCAI, T., AND GROT, B. Analyzing tail latency in serverless clouds with stellar. In *2021 IEEE International Symposium on Workload Characterization (IISWC)* (2021), IEEE, pp. 51–62.

[55] WANG, L., LI, M., ZHANG, Y., RISTENPART, T., AND SWIFT, M. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 133–146.

[56] WEYL, E. G., OHLHAVER, P., AND BUTERIN, V. Decentralized society: Finding web3's soul. *Available at SSRN 4105763* (2022).

[57] ZAMANI, M., MOVAHEDI, M., AND RAYKOVA, M. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (2018), pp. 931–948.

[58] ZHANG, T., XIE, D., LI, F., AND STUTSMAN, R. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 1–12.

A Appendix – Scheduler Analysis

Notation For a vector $\mathbf{v}$ we write $\mathbf{v}_j$ for the j'th entry in $\mathbf{v}$. By overloading notation, we write $\mathbf{e}_j$ for the unit vector with 1 on the j'th position and 0 everywhere else. We write $|\mathbf{v}|$ for $\sum_j \mathbf{v}_j$ – for vectors with positive entries $|\cdot|$ corresponds to $|\cdot|_1$.

For a set S we write $|S|$ for its size. For a real number x we write $|x|$ for its absolute value. We write $x \leftarrow a$ for assigning to variable x value a. If S is a set, we write $x \leftarrow S$ for a deterministic way of assigning a value $s \in S$ to variable x.

Problem statement An allocation for t canisters is represented by a vector $\mathbf{a} = (a_1, a_2, \ldots, a_t)$, a vector in $\{1,2,\ldots,v\}^t$ for some v. Given an allocation vector $\mathbf{a} = (a_1, a_2, \ldots, a_t)$, we define a deterministic (stateful) scheduling algorithm which, for each round $k \in \mathbb{N}$ outputs some index $sch(k) \in \{1,2,\ldots,t\}$, or equivalently, some unit vector $\mathbf{e}_{j^*}$ (with $j^* \in \{1,2,\ldots,t\}$)[1].

Intuitively, a good scheduler approximates the allocation vector well, i.e. for large enough k it outputs j a number of times proportional to its allocation. We formalize this intuition as follows.

[1] To simplify notation, we ignore that the scheduler is stateful

For each $k \in \mathbb{N}$ and $i \in \{1,2,\ldots,t\}$ let $idxs(k,i) = \{j \mid j \leq k, sch(j) = i\}$ the set of indexes j such that i was scheduled in round j (i.e. $s(j) = i$). For a good scheduler, the quantity

$$\left| \frac{|idxs(k,i)|}{k} - \frac{a_i}{|a|} \right|$$

is "small" for all large enough k. Informally, the above relation states that the scheduler has allocated k rounds proportional to the desired allocation.

Scheduler description Given an allocation vector $\mathbf{a}$, we define the following scheduler. The state of the scheduler at step k is given by vectors $\mathbf{d}(k), \mathbf{p}(k), \mathbf{s}(k)$ defined as follows[2].

The initial state is $\mathbf{d}(0) = \mathbf{s}(0) = (0,0,\ldots,0)$. For $k \geq 1$ define:

$$\begin{cases} \mathbf{p}(k) = \mathbf{d}(k-1) + \mathbf{a} \\ j^* \leftarrow \{j \mid \mathbf{p}_j(k) \geq \mathbf{p}_l(k),\ \forall l \in \{1,2,\ldots,t\}\} \\ \mathbf{s}(k) = \mathbf{s}(k-1) + \mathbf{e}_{j^*}, \\ \mathbf{d}(k) = \mathbf{p}(k) - \mathbf{e}_{j^*} \cdot |\mathbf{a}| \\ sch(k) = \mathbf{e}_{j^*} \end{cases}$$

Analysis The following lemma states some invariants that hold throughout the execution of the scheduler.

Lemma A.1. *For any $k \in \mathbb{N}$ it holds that:*

$$|\mathbf{d}(k)| = 0$$

For any $k \in \mathbb{N}^*$ it holds that:

$$|\mathbf{p}| = |\mathbf{a}|$$

Proof. We prove the invariant holds true for $\mathbf{d}$ by induction on k. The invariant for $\mathbf{p}$ follows immediately.

Base case For $k = 0$ we have that $\mathbf{d}(0) = (0,0,\ldots,0)$ so the invariant holds trivially.

Induction step By definition,

$$\mathbf{d}(k) = \mathbf{d}(k-1) + \mathbf{a} - \mathbf{e}_{j^*} \cdot |\mathbf{a}|$$

for some $j^* \in \{1,2,\ldots,t\}$. We then get that:

$$|\mathbf{d}(k)| = \left|\mathbf{d}(k-1) + \mathbf{a} - \mathbf{e}_{j^*}\right| = 0 + |\mathbf{a}| - |\mathbf{a}| = 0$$

□

The following lemma establishes a relation between $\mathbf{d}$ and $\mathbf{s}$. Informally, the relation says that $\mathbf{s}(k)$ is an approximation of $k \cdot \frac{\mathbf{a}}{|\mathbf{a}|}$ – the quality of the approximation is given by the entries in $\mathbf{d}$.

Lemma A.2. *For any $k \in \mathbb{N}$ it holds that:*

$$\mathbf{d}(k) = k \cdot \mathbf{a} - \mathbf{s}(k) \cdot |\mathbf{a}|$$

Proof. Proof by induction.

[2] In fact $\mathbf{p}$ is explicitly maintained only for convenience of analysis; it can be reconstructed from $\mathbf{d}, \mathbf{s}$ and $\mathbf{a}$

Base case For $k = 0$ we have that $\mathbf{d}(k) = \mathbf{s}(k) = (0, 0, \ldots, 0)$ so the equality holds trivially.

Induction step Assume it holds for $k-1$, i.e. $\mathbf{d}(k-1) = (k-1) \cdot \mathbf{a} - \mathbf{s}(k-1) \cdot |\mathbf{a}|$. By definition, $\mathbf{d}(k) = \mathbf{d}(k-1) + \mathbf{a} - \mathbf{e}_{j^*} \cdot |\mathbf{a}|$ for some j^*. From the induction step, this can be rewritten as

$$\begin{aligned} \mathbf{d}(k) &= (k-1) \cdot \mathbf{a} - \mathbf{s}(k-1) \cdot |\mathbf{a}| + \mathbf{a} - \mathbf{e}_{j^*} \cdot |\mathbf{a}| \\ &= k \cdot \mathbf{a} - \mathbf{s}(k) \cdot |\mathbf{a}| \end{aligned}$$

□

The following lemma establishes a lower bound on the the debt which processes can accumulate throughout their lifetime.

Lemma A.3. *For any $k \in \mathbb{N}$ and $j \in \{1, 2, \ldots, t\}$ it holds that:*

$$1 - |\mathbf{a}| \leq \mathbf{d}_j(k)$$

Proof. Proof by induction.

Inductive step The inequality trivially holds for $j \neq j^*$ since by definition (and the induction hypothesis):

$$\mathbf{d}_j(k) = \mathbf{d}_j(k-1) + \mathbf{a}_j \geq \mathbf{d}_j(k-1) \geq 1 - |\mathbf{a}|$$

To prove the bound for j^*, notice that by Lemma A.1, for any $k \in \mathbb{N}$:

$$\sum_{j=1}^{t} \mathbf{d}_j(k) + \mathbf{a}_j(k) = |\mathbf{a}|$$

Since j^* is such that $\mathbf{d}_{j^*}(k-1) + \mathbf{a}_{j^*} \geq \mathbf{d}_j(k-1) + \mathbf{a}_j$ for all j and $t \leq |\mathbf{a}|$, it holds that

$$\mathbf{d}_{j^*}(k-1) + \mathbf{a}_{j^*} \geq \frac{|\mathbf{a}|}{t} \geq 1.$$

So, we have that

$$\begin{aligned} \mathbf{d}_{j^*}(k) &= \mathbf{d}_{j^*}(k-1) + \mathbf{a}_{j^*} - (\mathbf{e}_{j^*} \cdot |\mathbf{a}|)_{j^*} \\ &\geq 1 - |\mathbf{a}| \end{aligned}$$

□

Finally, the following theorem establishes that in $|\mathbf{a}|$ rounds, job i is scheduled $\mathbf{a}_i$ times.

Theorem A.4.

$$\mathbf{s}(|\mathbf{a}|) = \mathbf{a}$$

Proof. By Lemma A.2, it holds that:

$$\mathbf{d}(|\mathbf{a}|) = |\mathbf{a}| \cdot \mathbf{a} - \mathbf{s}(|\mathbf{a}|) \cdot |\mathbf{a}| = (\mathbf{a} - \mathbf{s}(|\mathbf{a}|)) \cdot |\mathbf{a}|$$

Since $\mathbf{d}_{j^*} \geq 1 - |\mathbf{a}|$ (by Lemma A.3) and $\mathbf{d}_j(|\mathbf{a}|)$ is an integer divisible by $|\mathbf{a}|$ (by the above equality) then, it holds that $\mathbf{d}_j(k) \geq 0$. Since $|\mathbf{d}(|\mathbf{a}|)| = 0$ then $\mathbf{d}(|\mathbf{a}|) = (0, 0, \ldots, 0)$ and the desired equality follows.

□

PINOLO: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis

Zongyin Hao[1], Quanfeng Huang[1], Chengpeng Wang[2], Jianfeng Wang[3], Yushan Zhang[4], Rongxin Wu[1]*, Charles Zhang[2]

[1]School of Informatics, Xiamen University, [2]The Hong Kong University of Science and Technology,
[3]University of Southern California, [4]Tencent Inc.

{haozongyin, huangquanfeng}@stu.xmu.edu.cn, {cwangch, charlesz}@cse.ust.hk,
jianfenw@usc.edu, wurongxin@xmu.edu.cn

Abstract

DBMSs (Database Management Systems) are essential in modern enterprise software. Thus, ensuring the correctness of DBMSs is critical for enterprise applications. Among various kinds of bugs, logical bugs, which make a DBMS return an incorrect result set for a given SQL query, are the most challenging for detection since they typically do not result in apparent manifestations (e.g., crashes) and are likely to go unnoticed by users. The key challenge of detecting logical bugs is the test oracle problem, i.e., how to automatically characterize the expected results for a given query. The state-of-the-art approaches focus on generating the equivalent forms of queries via the customized rules, which rewrite a seed query to achieve the equivalent transformation. This dramatically limits the forms of SQL queries fed to the DBMS and thus leads to the under-reporting of many deeply-hidden logical bugs. In this paper, we propose a novel approach, PINOLO, to constructing a test oracle for logical bugs. Instead of generating the equivalent mutants of a seed query, our idea is to synthesize the queries that theoretically should return a superset or a subset of the result set of the seed query, forming the over-approximations or under-approximations of the seed query. A logical bug is detected if the result set returned by our synthesized query does not follow the expected approximation relation. We implemented our idea as a DBMS testing system and evaluated it on four widely-used DBMSs: MySQL, MariaDB, TiDB, and OceanBase. By the time of writing, PINOLO has found 41 unique logical bugs in these DBMSs, 39 of which have been confirmed by developers.

1 Introduction

Database Management Systems (DBMSs) are widely used as a key component in modern enterprise software. Their correctness and reliability are critical for many enterprise applications, such as online banking, e-shopping, e-payment, etc. Therefore, DBMS testing has attracted considerable attention in the industry [14, 23, 38, 41] and academia [10, 34–36]. For example, fuzzing, a widely-used testing technique, has been extensively applied to DBMSs [14, 38], showing its effectiveness in detecting crash bugs. However, as another typical kind of bug, logical bugs would cause DBMSs to return an incorrect result set for a given query but can easily go unnoticed by developers since they would not behave with apparent manifestations like system crash.

The predominant approach to detecting logical bugs consists of various automatic testing techniques. However, designing effective automatic testing techniques is non-trivial. One of the fundamental technical challenges is to characterize a correct result concerning a given query for comparison, which is a classical problem in testing, i.e., *test oracle problem* [12]. To tackle this challenge, researchers have proposed various ways to obtain the test oracle. The first category is based on differential testing [39]. It provides the same generated SQL query to multiple DBMSs for execution and resorts to the querying results to construct the test oracle. More concretely, the inconsistency among the result sets returned by different DBMSs indicates the presence of a potential logical bug. However, as pointed out by the existing studies [34, 36, 39], differential testing cannot be applied when a generated SQL query cannot comply with the grammar of all selected DBMSs or contains operations that have different semantics between different DBMSs. Although all the DBMSs support the common core syntax of SQL, each of them provides various extensions and forms its own dialect [39], which dramatically limits the generality of differential testing.

The second category is the oracle-guided synthesis approach [36], which does not rely on multiple DBMSs and thus mitigates the limitation of differential testing. It first specifies a randomly-selected row in a database table, namely, *pivot row*, as the test oracle and then synthesizes the query whose result set should contain this pivot row. The failure of fetching the row with the synthesized query evidences a potential bug underlying the tested DBMS. However, since such an approach considers only one row each time and the synthesis merely focuses on the *where* clause generation, it would miss logical bugs in various scenarios [34, 35]. For example, those rows that are duplicated to the pivot row are wrongly fetched or omitted, or the values processed by performing operators on the original row data are mistakenly computed and returned. Moreover, as pointed out by some recent studies [34, 35], the synthesis requires domain knowledge of the database dialect's supported operators and functions, and thus the implementation effort is high.

*Corresponding author: Rongxin Wu (wurongxin@xmu.edu.cn)

The third category is the metamorphic testing based approach [34, 35]. It first transforms a given query q into another query q' such that their querying results satisfy a specific relation, which is referred to as a *metamorphic relation*. The violation of the metamorphic relation upon the querying results indicates the wrong result of evaluating q or q'. For example, TLP [35] decomposes a query q into three partitioning sub-queries, each of which computes the result sets for a boolean predicate to be evaluated as **TRUE**, **FALSE**, and **NULL**, respectively, and then constructs an equivalent query q' by performing the union operation on these three sub-queries. NoREC [34] transforms an optimized version of a query into a non-optimized one by the customized rule, e.g., changing "**SELECT** * **FROM** t **WHERE** p" into "**SELECT** (p **IS TRUE**) **FROM** t." Compared with the aforementioned two categories of approaches, metamorphic testing based approaches are much more lightweight to implement and have been proven to be more effective in detecting logical bugs [34, 35]. However, existing studies instantiate the metamorphic relations as equivalent relations, which are still insufficient to detect many deeply-hidden bugs. This is because it is highly possible that, owing to the limited search space of mutations, the pair of equivalent queries would still share common buggy operators and functions and eventually return the same results. In such a case, the oracle from the equivalent query cannot provide any hint to detect logical bugs.

In this work, we present a new metamorphic testing based approach, named PINOLO, to detect logical bugs. Our idea to instantiate the metamorphic relation originates from the observation that the querying result of a given query is essentially a multi-set of tuples. The inclusion relation between the multi-sets, which is the foundation of set theory, is a good choice to characterize the metamorphic relation of two queries. Therefore, we try to mutate a given seed query to obtain the queries over or under-approximating it, of which the querying results are the superset or the subset of the one of the seed query. Based on the approximation relations, we can reveal a logical bug if the actual results violate the approximation relation. To systematically synthesize the two kinds of mutants, we introduce a series of approximate mutators, e.g., strengthening or weakening the predicates in *where* clauses, and propose an approximate query synthesis algorithm to generate the queries that have the over and under-approximation relations with the seed queries. Benefiting from our approximation relation and flexible approximate mutators, PINOLO can seize more opportunities to reveal logical bugs, as it can perform more aggressive mutations over the seed queries (e.g., discarding several functions), which explore the mutants of a seed query thoroughly. We also prove the correctness of our test oracle to solidify the theoretical foundation of PINOLO.

We implemented our idea as a DBMS testing system and evaluated it using four widely-used and comprehensively tested DBMSs, including MySQL, MariaDB, TiDB, and OceanBase. Compared with the state-of-the-art approaches, PINOLO is more effective in detecting logical bugs. During the 24-hour run, PINOLO can find 41 unique logical bugs, while the three state-of-the-art approaches together can only discover 14 bugs. Upon the submission, 39 out of 41 bugs have been confirmed by developers, showing the great impact of PINOLO on the four real-world DBMSs. In summary, this paper makes the following contributions:

- We introduce the concept of the approximation relation and a series of approximate mutators to resolve the test oracle problem in testing logical bugs in DBMSs.
- We propose a systematic metamorphic testing based approach PINOLO to detecting logical bugs in DBMSs, which leverages the approximate mutators to synthesize approximate queries for a seed query.
- We implement our idea as a DBMS testing system and systematically evaluate it using four widely-used DBMSs. The evaluation results demonstrate the effectiveness of PINOLO in detecting logical bugs.

2 Background

As discussed in § 1, this paper focuses on finding logical bugs in the DBMSs. This section provides several preliminaries as the background, including the concept of the DBMS, the logical bugs in the DBMSs, and the metamorphic testing based approaches for DBMS testing.

2.1 Database Management Systems

The DBMSs are widely used in many modern software systems. They enable the developers to perform various data manipulations, namely insertion, removal, update and search, according to their demands. Here, we concentrate on the relational DBMSs in our work as our target, which are one typical kind of DBMSs. Basically, they are developed on top of the relational model (RM) [7], where data is organized as a collection of tables. Each table is essentially a relation storing the records inserted by the developers, which is a multi-set of tuples from a mathematical perspective. Finally, a database in the DBMS consists of one or more tables, storing the data in a relational manner. In this paper, we use the DBMSs to indicate the relational DBMSs without introducing ambiguity.

There have been an increasing number of DBMSs released by the industry and academia, including MySQL, MariaDB, TiDB, and OceanBase [9, 22, 26, 29]. To interact with DBMSs, SQL [3], which is the most commonly used domain-specific language to store and operate data, was proposed. When retrieving data, developers write SQL queries and send them to DBMSs to get querying results. Each querying result is a multi-set of tuples indicating specific attributes of queried records in the tables. Overall, DBMSs provide an intuitive and flexible way to store and retrieve information, promoting the prosperity of database-backed applications in the real world.

```
--create a table
CREATE TABLE t ( c1 FLOAT );
INSERT INTO t VALUES (-1);

-- queries
(SELECT 1 FROM t WHERE COT(0.2)=0)
UNION ALL (SELECT (BINARY c1 | 0) FROM t);
--result: {0}

(SELECT 1 FROM t WHERE TRUE)
UNION ALL (SELECT (BINARY c1 | 0) FROM t);
--result: {18446744074709551615, 1}

(SELECT 1 FROM t WHERE FALSE)
UNION ALL (SELECT (BINARY c1 | 0) FROM t);
--result: {18446744074709551615}
```

Figure 1: An example of a logical bug in OceanBase.

2.2 Logical Bugs in DBMSs

With the prevalent usage of DBMSs in real-world industrial scenarios, their reliability and correctness have recently been paid increasingly more attention. As complex software systems, DBMSs can have bugs that cause crashes and other unexpected behaviors. Remarkably, the logical bugs are one of the most tricky bugs underlying the DBMSs. When a developer writes a SQL query and executes it upon a database, the returned result may be erroneous, which means that the semantics of the query is not correctly evaluated.

Figure 1 shows a logical bug in OceanBase [25]. We simplify the creation of the table for better demonstration. Specifically, the first query selects the constant value 1 from the table *t* if the cotangent of 0.2 is equal to 0, while the second and the third queries replace the predicates in the where clauses with 1 and 0, respectively. Obviously, the querying result of the first query should be a subset of the one of the second query, and meanwhile, it subsumes the querying result of the third one. However, the actual querying results do not conform to such inclusion relations. As confirmed by the developers of OceanBase, the querying result of the first query is incorrect, which is caused by the simultaneous usage of the set operator **UNION ALL** and the functions **COT** and **BINARY**.

As shown by the above example, logical bugs are more mysterious than system crashes, which have apparent manifestations. In contrast, people are often unaware of logical bugs in DBMSs. Typically, the developers of database-backed applications may realize the abnormal data retrieved from the database, while they are still uncertain whether their application is wrongly implemented or a logical bug of the DBMS is triggered. Therefore, detecting a logical bug in the DBMS has been typically recognized as a problem with both high impact and significant technical challenges.

2.3 Metamorphic Testing

Recent years have witnessed tremendous efforts in resolving the test oracle for logical bug detection in the DBMSs. Notably, the metamorphic testing based approach has been recognized to be state-of-the-art in DBMS testing for logical bug detection [35, 37]. Generally, the metamorphic testing based techniques attempt to construct multiple SQL queries of which the querying results have a specific relation, namely a metamorphic relation. If the querying results violate the metamorphic relation, we can have the confidence that at least one of the queries triggers a logical bug in the tested DBMS. For example, NoREC [34] transforms a query into a form in which the DBMS does not apply optimizations, which yields the test oracle that the two queries should make the tested DBMS return the same result. Besides, TLP [35] gets the equivalent query result by splitting the input query into several sub-queries and merging the results of sub-queries into one. When adapting the metamorphic testing, they take randomly-generated queries as the seed queries and then transform them into other queries to ensure the metamorphic relations, which automates the testing process for logical bug detection.

Unfortunately, the existing effort has not resolved the test oracle problem perfectly. To the best of our knowledge, previous studies only leverage the equivalent transformations, which are supported by the tested DBMS or conducted by their approaches, leaving the functions and operators in the query unchanged. This greatly limits their approaches to exploring the code of the tested DBMSs and thus reduces the chance of finding logical bugs. For example, the logical bug shown in Figure 1 can not be revealed by NoREC [34] and TLP [35], as the transformations preserve all the operators and the functions, still triggering the buggy evaluation process. To improve the capability of the metamorphic testing in finding logical bugs in the DBMSs, we propose a new DBMS testing approach in this work, which relaxes the equivalence relation with a less restrictive metamorphic relation, finally supporting finding more insightful logical bugs.

3 Approximate Query Synthesis

We propose the approximate query synthesis technique PINOLO[1] for detecting logical bugs in the DBMSs. Basically, our insight comes from the intuition that the mutation of specific grammatical constructs can induce the approximation relation between the original query and the mutated one, which can be adopted as the oracle of DBMS testing. Specifically, we start from a randomly generated seed query and mutate several constructs, such as predicates in where clauses, comparison operators, and set operators. Based on the mutation upon any seed query, we can successfully synthesize

[1] Pinolo is the English name of a cartoon character named Pinocchio. Once he tells a lie, his nose will become longer. The relative change in its length is the evidence to verify whether he is lying.

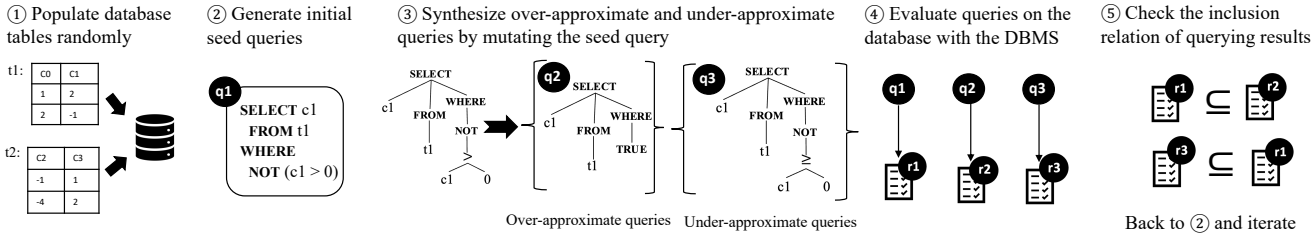


Figure 2: The workflow of PINOLO

a series of queries, of which the return results are the superset or the subset of the result of the seed query. If the seed query and a synthesized query do not produce the results with the expected inclusion relation, we safely conclude the existence of a bug underlying the DBMS. This section presents the system design of PINOLO to show how it resolves the oracle problem in detecting logical bugs in the DBMS and demonstrates the details of mutation-based query synthesis.

3.1 Approach Overview

We demonstrate the overall workflow of PINOLO in Figure 2. In the pre-processing phase, we populate several tables in a database by generating table records randomly, which leverages the existing DBMS random testing technique [27]. After preparing the database, PINOLO first generates a syntactically valid SQL query as the seed query. Then it parses the seed query and traverses its AST to determine whether each grammatical construct can be mutated. The mutations can make PINOLO synthesize several new queries of which the querying results have the inclusion relation with the one of the seed query, achieving the over or under-approximation for the seed query. Based on the synthesized queries and their approximation relation with the seed query, we further evaluate them on the populated database. Any violation of the approximation relation reveals a potential logical bug of the DBMS as at least one of the querying results of the seed query and the synthesized one is incorrect.

The critical component of PINOLO is to automatically generate the pairs of SQL queries with known approximation relations. To show more technical details, we first propose the approximation relation for SQL queries (§ 3.2), and then demonstrate how to synthesize the queries with approximation relations based on mutations (§ 3.3 and § 3.4). We summarize our design and highlight the advantage of PINOLO (§ 3.5).

3.2 SQL Query Approximation

In this work, we concentrate on the syntax of SQL queries shown in Figure 3. Basically, a SQL query can be a select-from-where query or the result of the set operation upon subqueries. The logical connectives and arithmetic operators make the query support depicting sophisticated predicates and

$$
\begin{aligned}
\textit{Relation } R &:= t \mid q \mid r_1 \otimes r_2 \mid r_1 \textbf{ JOIN } r_2 \\
\textit{Query } Q &:= q_1 \otimes q_2 \mid \textbf{SELECT } a \textbf{ FROM } r\ (\textbf{WHERE } p)^? \mid \\
&\qquad \textbf{SELECT DISTINCT } a \textbf{ FROM } r\ (\textbf{WHERE } p)^? \\
\textit{Pred } P &:= \ell_b \mid e_c \mid p \textbf{ IS } \ell_b \mid p \textbf{ IS NOT } \ell_b \\
&\qquad \mid p_1 \textbf{ AND } p_2 \mid p_1 \textbf{ OR } p_2 \mid \textbf{NOT } p \\
\textit{CExpr } E_c &:= e_{a1} \odot e_{a2} \mid e_a \odot \textbf{ALL}(r) \mid e_a \odot \textbf{ANY}(r) \\
\textit{AExpr } E_a &:= \ell_n \mid c \mid e_{a1} \oplus e_{a2} \mid f(e_a) \\
\textit{SOp } \otimes &:= \textbf{UNION} \mid \textbf{UNION ALL} \mid \textbf{INTERSECT} \mid \textbf{MINUS} \\
\textit{COp } \odot &:= > \mid < \mid \geq \mid \leq \mid == \mid \neq \\
\textit{AOp } \oplus &:= + \mid - \mid \times \mid \div \mid \cdots \\
\textit{BLit } L_b &:= \textbf{TRUE} \mid \textbf{FALSE} \qquad \textit{NLit } L_n := \ell_n \\
\textit{Attrs } A &:= c \mid e_a \mid e_a\ a \\
\textit{Table } T &:= t \qquad \textit{Attr } C := c
\end{aligned}
$$

Figure 3: The syntax of SQL queries

quantities. Without the loss of generality, we only instantiate an arithmetic expression with an integer literal, an attribute, the result of an arithmetic operator, and the result of a SQL built-in function. Notably, we can also utilize the keywords **ALL** and **ANY** to support advanced comparison between an arithmetic expression and the numeric attributes.

To resolve the test oracle problem in the DBMS testing, we follow the spirit of metamorphic testing and propose the approximation relations among the SQL queries. In what follows, we first formulate the concept of the approximation relation and then characterize the form of the SQL queries that can have approximation relations with other queries.

Definition 3.1. (Approximation Relation) Given a database D, a SQL query q_1 is the over-approximation of q_2 over D, denoted by $q_2 \trianglelefteq_D q_1$, if and only if $R(q_2, D) \subseteq R(q_1, D)$. Here, $R(q, D)$ is the return result set of q upon the database D, which is essentially a multi-set. $\subseteq$ is the inclusion relation between two multi-sets. We say q_2 is the under-approximation of q_1 over D, denoted by $q_1 \trianglerighteq_D q_2$, if and only if $q_2 \trianglelefteq_D q_1$.

Intuitively, the approximation relation between two SQL queries indicates the inclusion relation of their querying results. If we construct a pair of SQL queries (q_1, q_2) such that

$q_1 \trianglelefteq_D q_2$ or $q_1 \trianglerighteq_D q_2$, we can utilize the approximation relation as an instantiation of the metamorphic relation, which serves as the test oracle for DBMS testing.

Example 3.1. Assume that we have a database $D = \{\text{t1}\}$, where the schema of t1 is (c1) and $\text{t1} = \{(-1),(0),(1)\}$. Consider the following three queries.

q_1 : **SELECT** c1 **FROM** t1 **WHERE NOT** $(\text{c1} > 0)$

q_2 : **SELECT** $c1$ **FROM** t1 **WHERE TRUE**

q_3 : **SELECT** c1 **FROM** t1 **WHERE NOT** $(\text{c1} \geq 0)$

The first query q_1 selects all the non-positive values of the attribute c1 of the table named t1. The second query q_2 selects all the values of the attribute c1. The third query q_3 selects all the values of the attribute c1 that are not large than or equal to 0. Obviously, their querying results, denoted by $R(q_1,D)$, $R(q_2,D)$, and $R(q_3,D)$, respectively, have the relation that $R(q_3,D) \subseteq R(q_1,D) \subseteq R(q_2,D)$, implying $q_3 \trianglelefteq_D q_1 \trianglelefteq_D q_2$, i.e., $q_2 \trianglerighteq_D q_1 \trianglerighteq_D q_3$.

To sum up, the syntax shown in Figure 3 characterizes the search space of constructing a pair of queries with an approximation relation. Given a seed query in the syntax, we can always obtain a query upon a smaller/larger relation or with a stronger/weaker where clause, which induces a subset/superset of the return result of the seed query, achieving the under/over-approximation of the given seed query. Therefore, it is feasible to automatically generate the queries that have the approximation relation with a specific query q by mutating the query q, which trims/enlarges the relation or strengthens/weakens the predicate in the seed query. In this way, we can resolve the test oracle problem by synthesizing queries with approximation relations.

3.3 Approximate Mutators

Based on the key insight in § 3.2, we propose to resolve the test oracle problem by constructing SQL queries with the approximation relation. Specifically, we can always obtain the approximation relation if we transform a query to another one preserving the set inclusion relation of the relations and the implication relation of the predicates. According to high-level intuition, we propose the concept of the approximate mutator as follows, which is the fundamental ingredient of the approximate query synthesis in § 3.4.

Definition 3.2. (Approximate Mutator) An approximate mutator is a mapping from a SQL query q_1 to a query q_2 such that $q_1 \trianglelefteq_D q_2$ or $q_1 \trianglerighteq_D q_2$.

Essentially, an approximate mutator transforms a SQL query into another such that they have the over or under-approximation relation. We notice that a relation can be derived from other relations, e.g., the results of a select-from-where query and set operations, while compound and atomic logical expressions pose restrictions over relations. Therefore, we propose three categories of approximate mutators, which are shown in Table 1.

Table 1: Some representative approximate mutators. Transforming the construct **C1** into **C2** achieves the under-approximation, while transforming the construct **C2** into **C1** achieves the over-approximation.

Type	C1	C2
Relation	**SELECT** a **FROM** r	**SELECT DISTINCT** a **FROM** r
	r_1 **UNION ALL** r_2	r_1 **UNION** r_2
	r_1 **UNION** r_2	r_1
	r_1 **UNION** r_2	r_1 **INTERSECT** r_2
	r_1	r_1 **MINUS** r_2
Predicate	p	**FALSE**
	p **IS** ℓ_b	**FALSE**
	p **IS NOT** ℓ_b	**FALSE**
	TRUE	p
	TRUE	p **IS** ℓ_b
	TRUE	p **IS NOT** ℓ_b
Comparison expression	$e_{a1} \leq e_{a2}$	$e_{a1} = e_{a2}$
	$e_{a1} \geq e_{a2}$	$e_{a1} = e_{a2}$
	$e_{a1} \leq e_{a2}$	$e_{a1} < e_{a2}$
	$e_{a1} \geq e_{a2}$	$e_{a1} > e_{a2}$
	$e_{a1} \neq e_{a2}$	$e_{a1} < e_{a2}$
	$e_{a1} \neq e_{a2}$	$e_{a1} > e_{a2}$
	$e \odot$ **ANY**(r)	$e \odot$ **ALL**(r)

Concretely, the mutators alter the relations and predicates in a SQL query, and meanwhile, mutate the comparison expressions, which are often atomic constraints in a predicate, achieving the approximation relation between the queries before and after the mutation. For each row, if we replace the SQL grammatical construct in the second column with the one in the third column, we can obtain a query that under-approximates the original one; if we replace the one in the third column with the one in the second column, we can obtain a new query that over-approximates the original query. Now we provide more explanations on the approximate mutators.

- **Mutating relations.** Using **DISTINCT** in a select-from-where query removes the duplicate values in the querying result, which under-approximates the original query. The set operator **UNION ALL** preserves the duplicate values and the operator **UNION** does not, so replacing the former with the latter ensures the under-approximation relation between the new query and the original one. Other mutators altering relations are fairly simple.
- **Mutating predicates.** For an arbitrary predicate p, we can strengthen it by mutating it to **FALSE** and weaken it by mutating it to **TRUE**. The logical implication would pose more or less restrictive constrain upon the tuples in the relations, which finally achieve the under-approximation or over-approximation, respectively.
- **Mutating comparison expressions.** For each comparison expression as an atomic constraint, we can alter its comparison operator to strengthen or weaken the constraint induced by the expression. For example, replacing

$\geq$ with $=$ makes the new expression more restrictive than the original one. Also, mutating the keyword **ANY** with **ALL** also induces a stronger predicate in the query.

Example 3.2. For the simpler SQL query q_1 in Example 3.1, we can mutate it by replacing the negation with **TRUE** and altering $>$ to $\geq$, which yield two queries q_2 and q_3 that over-approximate q_1, respectively. We can further consider a more complex SQL query as follows.

$\widetilde{q}$: **SELECT** 1 **FROM** t1 **WHERE**
(**NOT** (**FROM_DAYS**(1) = **ALL**(**SELECT** c1 **FROM** t1)))

We can mutate the predicate in the where clause of $\widetilde{q}$ to **TRUE** to over-approximate the query $\widetilde{q}$. Also, mutating **ALL** to **ANY** weakens the comparison expression in the negation, and thus, strengthens the predicate in the where clause, yielding an under-approximation of the query $\widetilde{q}$.

Notably, the approximate mutator proposed in this section is a general concept. The mutators shown in Table 1 are just several instances of the mutators, while we can further define more mutators to enable us to obtain the queries with approximation relations more flexibly. Actually, we provide a systematic framework to instantiate such mutators in these categories. The complete list of the instantiated mutators, including REGEXP and IN operators, has been published online [31].

3.4 Mutation-based Query Synthesis

Leveraging the approximate mutators in § 3.3, we can finally propose the approximate query synthesis algorithm by applying the mutators upon a seed query. This section demonstrates the technical details of the synthesis algorithm on how to generate two sets of SQL queries that over-approximate and under-approximate a seed query, respectively. We also formulate our test oracle with a theorem as the theoretical guarantee for the approximation relations among the seed query and the synthesized ones.

Algorithm 1 shows the mutation-based query synthesis algorithm. Initially, it takes a seed query as the input, parses the query, and generates an AST of the query to facilitate further mutations (Line 2). Basically, it traverses the AST in a top-down manner, during which it identifies the potential SQL constructs for the mutation (Line 3–Line 4). Consider synthesizing the queries that under-approximate the seed query as an example, where *kind* is set to be *UNDER* (Line 4). Specifically, it processes each SQL construct in two ways.

- When encountering the SQL construct (r_1 **MINUS** r_2), it attempts to trim the relation r_1 and enlarge the relation r_2 so that the difference of the two relations can be trimmed (Line 12–Line 15). Similarly, it strengthens the predicate p for (**NOT** p) and (p **IS NOT TRUE**), and also enlarges the relation r for the comparison expression $e \odot$ **ALL**(r) (Line 16–Line 24).

Algorithm 1: Mutation-based query synthesis

```
1  Procedure synthesizeApproximateQueries(q):
2      τ ← parseQuery(q) ;
3      Q_over ← mutate(q, τ, OVER) ;
4      Q_under ← mutate(q, τ, UNDER) ;
5      return (q, Q_over, Q_under);
6  Procedure mutate(u, τ, kind):
7      if kind == OVER then
8          negKind ← UNDER;
9      else
10         negKind ← OVER;
11     S ← applyApproxMutator(q, τ, kind) ∪ {q};
12     if u : (r1 MINUS r2) then
13         S'1 ← mutate(r1, getAST(r1), kind);
14         S'2 ← mutate(r2, getAST(r2), negKind);
15         S ← S ∪ {r'1 MINUS r'2 | r'1 ∈ S'1, r'2 ∈ S'2} ;
16     else if u : (NOT p) then
17         S' ← mutate(p, getAST(p), negKind);
18         S ← S ∪ {NOT p' | p' ∈ S'};
19     else if u : (p IS NOT TRUE) then
20         S' ← mutate(p, getAST(p), negKind);
21         S ← S ∪ {p' IS NOT TRUE | p' ∈ S'};
22     else if u : e ⊙ ALL(r) then
23         S' ← mutate(r, getAST(r), negKind);
24         S ← S ∪ {e ⊙ ALL(r') | r' ∈ S'} ;
25     else
26         Γ ← getSubASTs(u) ;
27         Π ← ⊥;
28         foreach (v, τ_v) in Γ
29             Π[(v, τ_v)] ← mutate(v, τ_v, kind);
30         S ← S ∪ concat(q, Γ, Π) ;
31     return S;
```

- For other SQL constructs, it trims each relation and strengthens each predicate and comparison expression appearing in the constructs. Lastly, it composes each mutated constructs together by the function *concat* to obtain the ASTs of the synthesized queries under-approximating the seed query (Line 26–Line 30).

By applying the approximate mutators during the AST traversal, Algorithm 1 finally synthesizes two sets of queries on the fly, which are syntactically valid and have the approximation relation with the seed query q.

Example 3.3. Consider the query $\widetilde{q}$ in Example 3.2 as the seed query. We show how to synthesize the queries that under-approximate $\widetilde{q}$. After generating its AST, which is shown by the leftmost tree in Figure 4, Algorithm 1 examines each SQL construct in a top-down manner. When encountering the predicate in the where clause, we can mutate the predicate, which is a logical negation, to **FALSE**, or strengthen the predicate in the logical negation. For the latter case, we can further mutate the comparison expression in the negation to **TRUE**, mutate the comparison operator with $\geq$, or replace **ALL** with **ANY**, which finally weakens the logical negation. Finally, we can

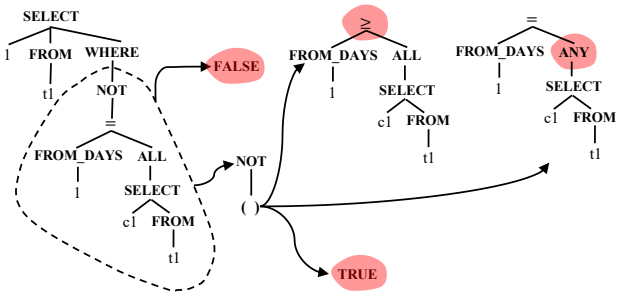


Figure 4: An example of synthesizing queries under-approximating the query $\widetilde{q}$

obtain four queries that under-approximate $\widehat{q}$. Particularly, one of the synthesized queries is as follows:

$\widetilde{q}'$: **SELECT** 1 **FROM** t1 **WHERE**
(**NOT** (**FROM_DAYS**(1) $\geq$ **ALL**(**SELECT** c1 **FROM** t1)))

Fortunately, we find that $R(\widetilde{q}', D)$ is not subsumed by $R(\widetilde{q}, D)$, indicating that the querying result of $\widetilde{q}$ or $\widetilde{q}'$ is incorrect, which is confirmed by the developers of MySQL [21].

It is worth mentioning that we have to restrict the values in the database tables not to be null. Any comparison between a non-null value and a null value can introduce an unknown value of the comparison expression, denoted by **NULL** without the ambiguity, which is smaller than **TRUE** but larger than **FALSE** in the logical order. However, we can notice that the predicate (p **IS NULL**) evaluates to **FALSE**, **TRUE**, or **FALSE**, if p is **TRUE**, **NULL**, or **FALSE**, respectively, which indicates that strengthening or weakening the predicate p does not always strengthen or weaken the predicate p **IS NULL**. In this case, we cannot ensure the expected approximation relation between the seed query and the synthesized queries in Q_{over} and Q_{under}. Formally, we state the following theorem to formulate our test oracle.

Theorem 3.1. (Test Oracle) Assume that the database D does not contain any table storing null values. Taking the query q as a seed query, Algorithm 1 can always synthesize two sets of queries Q_{over} and Q_{under} such that:

- For any $q' \in Q_{over}$, we have $q \trianglelefteq_D q'$, i.e., $q' \trianglerighteq_D q$.
- For any $q' \in Q_{under}$, we have $q' \trianglelefteq_D q$, i.e., $q \trianglerighteq_D q'$.

To solidify the theoretical foundation of our test oracle, we sketch the proof of the theorem briefly. First, the fact that the database tables do not contain null values implies that the evaluation results of any expressions, including comparison expressions and predicates, are not evaluated to unknown values, and the intermediate relations, such as the results of the join and the union operator, do not contain null values. Second, it is trivial to prove that the approximate mutators applied to different constructs finally yield a weaker predicate or larger relation when *kind* is *OVER* in the absence of null values. A similar argument also holds when *kind* is *UNDER*. Therefore, we can prove the approximation relation between each synthesized query and the seed query based on the principle of structural induction.

3.5 Summary

PINOLO automates the DBMS testing via the approximate query synthesis, which discovers underlying logical bugs in the DBMSs. The syntax in Figure 3 ensures the syntactical validity of the seed queries, and furthermore, guarantees that the synthesized queries have valid SQL syntax. Meanwhile, our approximate mutators enable us to obtain the approximation relation between the seed query and the synthesized ones, which perfectly resolves the test oracle problem. Compared with the existing techniques [34–36], PINOLO considers more SQL features, such as set operators, arithmetic expressions, sub-queries, etc. The expressive syntax permits us to test the DBMS more thoroughly and discover more logical bugs reported in previous studies, which will be evidenced by our experiments. Besides, the approximate operators can aggressively mutate the seed query, which may remove the buggy operators and functions, revealing the logical bugs more thoroughly than existing techniques. Lastly, it is worth noting that PINOLO provides a general framework for discovering the logical bugs in the DBMS. We can further extend the syntax of SQL queries and instantiate more approximate mutators, which can promote the capability of discovering the logical bugs triggered by sophisticated SQL queries.

4 Implementation

We implemented our approach PINOLO as a DBMS testing system, which was written in GO with 8,055 lines of code. The source code of our tool is hosted in the github repository [2]. Next, we present more details of our implementation decisions that are important for the outcome of our experiments.

Database population. We randomly generate the database tables by leveraging an existing tool, GO-RANDGEN [27]. Following the best practice summarized in the prior study [36], we restrict the number of table records to be no more than 30. Besides, we randomly generate the tables with the same attributes, which makes the join operator yield a non-trivial result. Particularly, we avoid null values in any table, as the test oracle requires non-null values as a prerequisite, which is stated in Theorem 3.1.

Seed query generation and parsing. To generate seed queries as the input of our synthesis algorithm, we utilize GO-RANDGEN [27] to automatically produce seed SQL queries. Specifically, we use the general-purpose parser generator BISON [13] to write a context-free grammar file describing the SQL syntax with a series of production rules. We provide

[2] https://github.com/qaqcatz/impomysql.git

this grammar file to GO-RANDGEN, so that it can generate the queries by searching each production rule randomly. It then heuristically selects the terminal or non-terminal symbols to avoid exceeding the limitation of recursion. Moreover, we permit users to write embedded LUA code blocks in the grammar file to further restrict the form of seed queries, which can ensure the successful query execution, e.g., the number of columns in the two queries of **UNION** should be equal. To apply the approximate mutators to seed queries, we utilize another tool PINGCAP PARSER [28], which accepts the same context-free grammar as the one for the seed query generation, to generate ASTs of seed queries for the mutation.

Approximate mutator instantiation. As mentioned at the end of § 3.3, we instantiate a series of approximate mutators for the relations, predicates, and comparison expressions in a given query. Each approximate mutator consists of two SQL grammatical constructs, which indicate the construct after the over and under-approximation, respectively. To cover most features of DBMSs, we instantiate 25 approximate mutators in total, including the approximate mutators demonstrated in Table 1. Among them, 5, 6, and 14 approximate mutators correspond to the mutations of the relations, predicates, and comparison expressions, respectively. Apart from the approxmiate mutator in Table 1, we also include 7 mutators supporting **LIKE**, **REGEXP**, **IN**, and **BETWEEN**.

Bug report post-processing. After synthesizing queries, PINOLO obtains the querying results by evaluating queries on the populated database in the tested DBMSs. It is noted that inconsistent querying results frequently occur in our testing process. For example, during a 24-hour testing period, 46,772 inconsistent query pairs are generated for MySQL. The large number of such query pairs makes the process of confirming and fixing bugs quite verbose. To make the testing results easier to understand, we borrow the idea of *delta debugging* [56] to localize the root cause of the inconsistent returned results of each query pair. Specifically, we associate each problematic query pair with a release version of the under-test DBMS's code base, the earliest one where inconsistent query results appear. We denote such release version with respect to a given query pair as the bug-inducing version. Further, two bugs are considered the same if their bug-inducing versions are the same. Based on our interactions with DBMS developers, our bug reports and the release version's change lists can help developers pinpoint culprit updates easily.

5 Evaluation

To evaluate the performance of our approach PINOLO in detecting logical bugs in the popular real-world DBMSs, we design the following research questions:

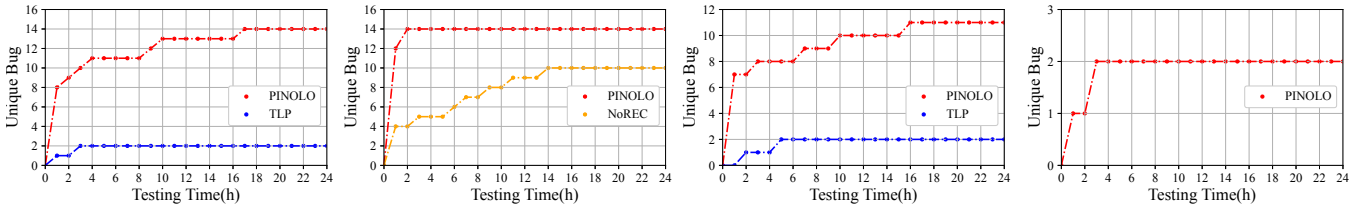
- **RQ1:** How many logical bugs in real-world DBMSs can be detected by PINOLO?
- **RQ2:** Can PINOLO outperform the state-of-the-art logical bug detection techniques?
- **RQ3:** How does the randomness introduced by the seed query generation affect the performance of PINOLO?

5.1 Experiment Setup

Environment. We conducted the experiments on one server with 104-cores Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz and 500 GB memory. The server runs Ubuntu 18.04 system that uses Linux kernel version: 5.4.0-135-generic. To ensure fair comparisons, we allocated four threads for each DBMS testing in our following experiments.

Tested DBMS. Our focus was on testing four widely-used and large-scale open-source DBMSs: MySQL, MariaDB, TiDB, and OceanBase. There are two main reasons for the subject selection. First, these selected DBMSs are representative DBMSs from phenomenal open-source and/or commercial products: MySQL and MariaDB are the two most well-known open-source DBMSs; OceanBase is a mature commercial database product from Ant Group; TiDB is developed by PingCap Inc. They are also commonly used in the evaluation of previous studies [34–36]. Moreover, we admit and discuss potential limitations introduced by our selection of DBMS systems in § 6. Second, we chose a DBMS whose SQL syntax is compatible with MySQL as an evaluation subject to reduce the implementation effort. This is because, although our approach can be generalized to other DBMSs, the implementation of the seed query generation and parsing (See § 4) requires a grammar file that describes the SQL syntax of a tested DBMS. To obtain timely feedback from developers, we tested the latest release versions of the selected DBMSs: MySQL 8.0.31, MariaDB 10.11.1, TiDB 6.4.0, and OceanBase 4.0.0.

Baseline. We compared PINOLO with the three state-of-the-art logical bug detection techniques, namely PQS [36], NOREC [34], and TLP [35], respectively, which correspond to three kinds of test oracles. Similar to our approach, these baselines also require knowledge about the SQL syntax of different DBMSs for seed query generation and parsing. Unfortunately, their implementations cannot support all the selected DBMSs. We also sought help from their authors, but they still could not fix the problems before the paper submission. Therefore, we skipped the evaluation of the baselines on the unsupported DBMSs. Moreover, we tried to use the same random seed as the baselines. However, we found that they can neither export their random seeds nor import the random seeds provided by users. Therefore, we generated the random seeds by ourselves for PINOLO. To understand the impacts of the random seed query generation, we investigated how PINOLO is robust to such randomness in § 5.4.



(a) MySQL (b) MariaDB (c) TiDB (d) OceanBase

Figure 5: Comparison between PINOLO and the baselines. (a)-(d) show the number of unique logical bugs over time (24 hours) on each DBMS. We ignore the results of the baselines on the unsupported DBMSs and the baselines finding no bugs.

Table 2: The demographics of the DBMSs under the test

DBMS	Version	GitHub Stars	LOC	First Release
MySQL	8.0.31	8.6K	4,766,086	1995
MariaDB	10.11.1	4.6K	3,727,410	2009
TiDB	6.4.0	33.1K	985,518	2017
OceanBase	4.0.0	5.1K	2,722,881	2021

Table 3: Applicability of existing logical detection techniques and PINOLO for the selected DBMSs

DBMS	PQS	NOREC	TLP	PINOLO
MySQL	✓	×	✓	✓
MariaDB	×	✓	×	✓
TiDB	×	×	✓	✓
OceanBase	✓	✓	✓	✓

5.2 Effectiveness of PINOLO

We used PINOLO to test the latest version of MySQL, MariaDB, TiDB, and OceanBase for 24 hours. Table 4 summarizes the results of PINOLO. The column **All** shows the number of problematic query pairs that induce unexpected results, which indicate the existence of logical bugs. As we can see, PINOLO discovered a large number of problematic query pairs, ranging from 4,675 to 46,772 for the tested DBMSs. However, we found that most of these pairs can be attributed to the same bug. To relieve the developers from the heavy burden of checking the duplicate bugs, we leverage the bug-inducing version to deduplicate the bugs (See § 4). The column **Unique** shows the number of bug reports after deduplication, which is significantly smaller than the value in column **All**, ranging from 2 to 14. We submitted the deduplicated bugs to the developers for confirmation. The column **Verified** shows the number of bug reports that have been verified by developers, ranging from 2 to 14.

Table 4: The number of logical bugs found by PINOLO.

DBMS	All	Unique	Verified
MySQL	46,772	14	14
MariaDB	42,208	14	12
TiDB	5,268	11	11
OceanBase	4,675	2	2

In total, PINOLO found 41 unique logical bugs in these DBMSs, 39 of which have been confirmed by developers. For MySQL, TiDB, and OceanBase, all of the detected bugs have been confirmed by developers. For MariaDB, twelve out of fourteen bugs have been confirmed, while the rest are still waiting for the investigation. We sampled several bug reports of MariaDB submitted by others and found that developers typically take a much longer time to handle the bugs. To keep track of the status of our reported bugs, we release the bug list in a public GitHub repository[3].

Answer to RQ1: PINOLO discovers 41 unique bugs on MySQL, MariaDB, TiDB, and OceanBase, 39 of which have been confirmed by DBMS developers.

5.3 Comparisons on Detecting Logical Bugs

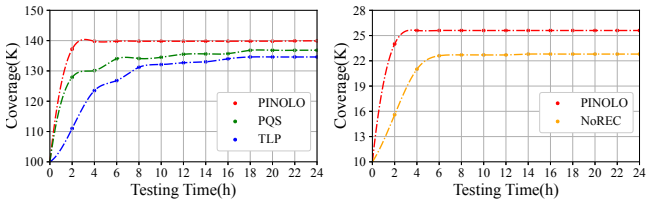
Logical bug detection. We compared PINOLO with the three state-of-the-art baselines, i.e., PQS, NOREC, and TLP. We ran all methods with a time budget of 24 hours. The comparison results are shown in Figure 5. Note that the baselines do not support all the DBMSs, and thus we only concentrated on the comparison between PINOLO and the runnable baselines with respect to each DBMS.

For MySQL, PINOLO detected 14 logical bugs, while TLP only discovered 2 bugs. For MariaDB, PINOLO detected 14 bugs, while NOREC discovered 10 bugs. For TiDB, PINOLO detected 11 bugs, while TLP only discovered 2 bugs. For OceanBase, PINOLO can detect 2 bugs, while the baselines cannot find any. We also manually verified the overlap in the bugs detected by PINOLO and the baselines. For MariaDB, 4 out of 10 bugs detected by NOREC can also be found by PINOLO. For TiDB, 1 out of 2 bugs detected by TLP can also be found by PINOLO. There are no bugs detected by PQS.

Figure 5 shows the logical bug detection progress over time for PINOLO and the baselines. We found that PINOLO is more efficient in finding logical bugs compared to all of the baselines. Within one hour, PINOLO was able to detect 57.1% (8/14) of bugs on MySQL, 85.7% (12/14) on MariaDB, 63.6% (7/11) on TiDB and 50% (1/2) on OceanBase.

Code coverage. To understand why PINOLO can find more

[3]https://github.com/qaqcatz/impomysql_bugreports



(a) MySQL (b) MariaDB

Figure 6: Code coverage comparison between PINOLO and the three baselines over 24 hours

logical bugs than all other methods, we used GCOV [30] to compute the line coverage achieved by PINOLO and all others. Note that we are unable to provide code coverage results for TiDB and OceanBase. As TiDB is developed in GO, we cannot find any feasible tool to support the program instrumentation or the code coverage profile for system test [4]. For OceanBase, the instrumented binary has to be deployed by the specific tool OBDEPLOY [24], which breaks the functionality of code coverage profiling.

Figure 6 shows the code coverage over time for MySQL and MariaDB. The results show that PINOLO achieves a higher line coverage than the three baselines. For MySQL, the improvement of PINOLO over PQS and TLP is 2.2% (3,008 lines) and 4.0% (5,316 lines), respectively. For MariaDB, the improvement of PINOLO over NOREC is 12.4% (2,835 lines).

Code coverage is well recognized as an approximation of testing capability, because a bug cannot be detected if its buggy code is not executed. However, larger code coverage does not mean more bugs. To better understand the impact of larger code coverage achieved by PINOLO, we further investigated whether there were some bugs whose buggy code is uniquely covered by PINOLO. TiDB#40015 [42] is a typical example. The root cause of this bug is the improper exception handling in the function *vecGetDateFromString*, which has been covered by PINOLO but missed by other baselines during the 24-hour running.

Answer to RQ2: Compared with the state-of-the-art techniques, PINOLO can find more unique logical bugs and achieve higher line coverage.

5.4 Impacts of The Seed Query Generation

PINOLO uses GO-RANDGEN, which takes a random seed to generate a set of seed queries. To understand whether the randomness affects the efficiency and effectiveness of PINOLO, we conducted the experiments which used GO-RANDGEN with five different random seeds to generate five sets of seed queries. We then ran PINOLO under each set and compared their performance on detecting logical bugs.

[4] Go 1.20 plans to cover the features of program instrumentation and code coverage profile, but will be available after Feb 2023 [15].

Table 5: The numbers of the discovered logical bugs when feeding different seed queries to PINOLO

DBMS	Seed1	Seed2	Seed3	Seed4	Seed5	Common
MySQL	14	18	16	16	17	11
MariaDB	14	16	13	13	13	10
TiDB	11	9	11	11	13	9
OceanBase	2	2	2	2	2	2

Table 6: Importance of the logical bugs found by PINOLO

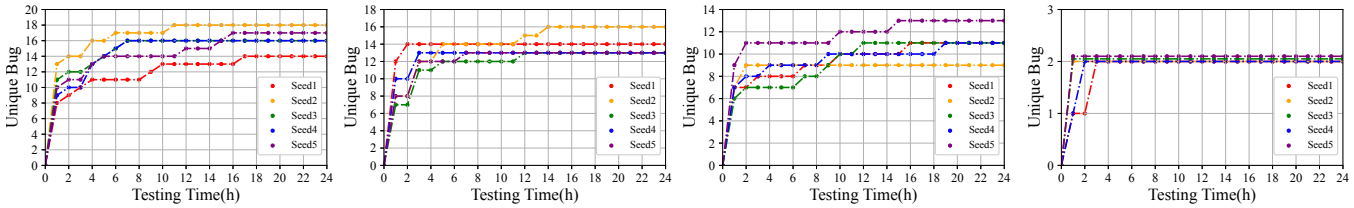
DBMS	Severity		Bug impact duration		
	S2	S3	<1 year	1∼5 years	5∼10 years
MySQL	6	8	1	7	6
MariaDB	9	3	1	7	4
TiDB	8	3	5	6	0
OceanBase	-	-	1	1	0

Figure 7 shows the progress of detecting unique logical bugs over time for the five sets of seed queries. We found that the growth trend of the number of unique bugs over time is similar under different sets. Table 5 shows more details about these detected logical bugs. Among the five sets of random seed queries, the common bugs, of which the numbers are shown in the column **Common**, account for an average of 68.4%, 72.9%, 82.9%, and 100% of the total bugs on MySQL, MariaDB, TiDB, and OceanBase respectively. This result shows that PINOLO can find different unique logical bugs via different sets of random seed queries.

Answer to RQ3: The randomness introduced by the seed query generation does not have a significant impact on the overall bug detection performance of PINOLO. Meanwhile, the different random seed queries can benefit PINOLO in detecting different bugs.

5.5 Discussion

Bug Importance. To understand the importance of the bugs found by PINOLO, we investigated their severity and the impact duration. The results are shown in Table 6. The column **Severity** indicates the severity of the bugs labeled by developers. Our reported bugs were classified as the levels of S2 and S3, which represent the second and third highest severity levels, respectively. Typically, the S2 level indicates a severe loss of service or missing significant functionality, while the S3 level indicates a minor loss of service or inconvenient usage. Note that there is no severity level in the bug tracking system of OceanBase, so we skipped the discussion for OceanBase. We discovered 6, 9, and 8 bugs in the S2 level in MySQL, MariaDB, and TiDB, respectively, which account for 62.2% of the bugs in the three DBMSs. The bugs in the S3 level are less severe, but developers still considered them to be fixed



(a) MySQL (b) MariaDB (c) TiDB (d) OceanBase

Figure 7: The number of unique logical bugs over time for each group of the random seed queries on the tested DBMSs

necessarily. For example, we received an appreciation from the MySQL developers in one of the bug reports with the S3 level: "*Thank you for your contribution. It is our standpoint that all bugs should be fixed, whether major or minor.*"

Table 6 also shows the **bug impact duration**, which is computed as the interval between the time of the bug-inducing version and the bug reporting time. Surprisingly, we found that 10 (25.6%) and 21 (53.8%) of bugs have lasted for 5-10 and 1-5 years, respectively. Specifically, the two earliest bugs of MySQL [20] and MariaDB [17] can be traced back to 2014. This result indicates that the logical bugs are typically difficult and slow to be found, which is also consistent with the findings of the prior study [35].

False Positive and False Negative. According to Theorem 3.1, PINOLO will not produce false positives in theory. However, we observe two bugs that have not been confirmed by developers for more than five months, and thus suspect that they are false positives. We manually inspected the two cases. In one bug report, a query returns "0", while the approximated query returns "-0". In the other one, a query returns "0", while the approximated query returns "-0.001". Although PINOLO considers the above inconsistency results as bugs, developers may have a higher tolerance for such inconsistencies. These reports allowed us to refine our implementation to reduce false positives in the future.

In terms of false negatives, we observed 9 cases that are detected by the baseline approaches but cannot be detected by PINOLO. This is mainly due to the DBMS features that are currently not supported by PINOLO. Specifically, there are six, two and one bugs that are related to aggregate functions, left/right join, and three-valued logic, respectively.

Limitations and Future Work. We currently do not support all features of DBMSs, such as aggregate functions, window functions, and left/right join. This is because, these features may break the approximation relation (Definition 3.1). For example, aggregate functions typically map a set of values into a single value (e.g., sum, average, maximum, minimum, and so on), and the mapped value will not preserve the inclusion relation of the original sets, thus breaking the approximation relation. In our future work, we will design new approximation relations to support more features. In addition, we intend to explore the application of metamorphic testing in other system software domains (such as networking and distributed systems [1, 32, 43, 44]).

As shown in § 5.4, different random seed queries can benefit PINOLO in detecting different bugs. This indicates that adjusting seeds dynamically is helpful to make PINOLO find more bugs. Therefore, in the future, we will consider to integrate PINOLO into the fuzzing framework to better prioritize the seed selection and enhance the bug detection capability.

Another possible direction for the future exploration is bug deduplication. In this work, we determine whether two bugs are duplicated by checking their bug-inducing versions. However, the bug-inducing version is an approximation of root causes, which would misclassify the bugs. This is because a release version of DBMS would introduce numerous bugs, which lead to multiple problematic query pairs. In the future, we will consider leveraging the spectrum-based fault localization techniques or mutation testing to improve the precision of discovering the root cause of the bugs, so as to improve the precision of bug deduplication.

6 Threats to Validity

The threat to internal validity is primarily associated with the implementation of our approach. To mitigate this concern, we have employed several DBMSs to cross-check whether the mutants generated by PINOLO accurately represented over-approximations or under-approximations of the seed query.

The threat to external validity lies in the representative of the evaluation subjects. Our proposed approach was evaluated on a restricted set of DBMSs, as explained in the evaluation section. As a result, the conclusion drawn in this paper may be limited. However, we believe that it is non-trivial to detect new bugs in these selected DBMSs, as they have been thoroughly tested by SQLancer [33]. In the future, we will extend the implementation of PINOLO to support the evaluation of additional open-source and commercial DBMSs.

7 Related Work

PINOLO is an unique DBMS testing system for finding logical bugs in DBMSs, but draws inspiration from several areas in the literature, including DBMS testing, metamorphic testing, differential testing, and grammar fuzzing.

DBMS testing. Recent efforts on DBMS testing focus on various aspects of DBMSs. Most of them target discovering logical bugs in the relational DBMS [34–36]. They use specific metamorphic relations or multiple implementations as the oracles and generate syntactically valid SQL queries for metamorphic testing [5] or differential testing [18]. Some also attempt to improve the coverage of the DBMSs and leverage fuzzing techniques to enumerate the queries in various forms [10, 46, 49, 57]. PINOLO uses a new design, termed approximate query synthesis, and does not use other DBMS implementations as the oracle.

Metamorphic testing. Metamorphic testing [5] has become more and more popular over the past decade. It has been used in testing many software systems, such as compilers [16, 50], SMT solvers [48, 54], DBMSs [35, 36], and AI systems [2, 47, 55]. Metamorphic testing uses one type of metamorphic relations to compare outputs produced by a seed input and a mutated one. As long as the two outputs violate the specific metamorphic relation, then at least one of the two inputs yields a wrong result [4]. PINOLO utilizes an instantiation of a metamorphic relation, i.e., the approximation relation, as an effective testing oracle for discovering logical bugs in the DBMSs. Similar to the skeleton approximation enumeration in the SMT solver testing [54], PINOLO performs the over-/under-approximation of the seed queries. However, PINOLO has to deal with more sophisticated syntax and supports more flexible mutations so that more buggy operators and functions can be removed. Therefore, it is able to detect insightful logical bugs that existing approaches, such as NOREC [34] and TLP [35], fail to discover.

Differential testing. Apart from metamorphic testing, differential testing provides another testing paradigm for resolving the oracle problem in software testing [6, 40, 52, 53]. Utilizing another software system with the same functionality as an oracle implementation, differential testing techniques compare the system's outputs under testing and the oracle implementation and reveal potential functional bugs with the divergent outputs. Although the techniques can generate the inputs of the systems flexibly, they can only be applied to software systems that have other implementations supporting the same functionality, such as JVM [6], ORM frameworks [40], and SMT solvers [53]. We believe PINOLO and other metamorphic testing approaches are orthogonal to differential testing techniques, which can be applied and strengthened to each other in testing DBMSs.

Grammar fuzzing. Grammar fuzzing is used to generate inputs that satisfy a specific language syntax [8, 11, 19, 19]. It has been widely used in testing many real-world software systems, such as browsers [45], compilers [51], and DBMSs [36, 57]. The generated inputs can always pass the syntax checking of software systems, which avoids the unnecessary enumeration of the inputs in an ill form, improving the effectiveness of generated inputs. Instead of relying on a specific grammar, PINOLO mutates an existing seed query to synthesize an approximate query. Our synthesis process rewrites specific grammatical constructs in the seed query, which ensures the syntactical validity of the synthesized one. We do think it is also promising to utilize existing grammar fuzzing techniques to enumerate initial seed queries automatically [36, 57], which can provide more opportunities for improving the coverage in the DBMS testing.

8 Conclusion

This paper presents PINOLO, an automatic query synthesizer for discovering logical bugs in DBMSs. Given a seed query, PINOLO mutates specific SQL constructs and generates a query that over-approximates or under-approximates the seed query. We posit that the approximation relation provides effective guidance for discovering logical bugs underlying DBMSs. Our experimental results demonstrate the effectiveness of PINOLO. Benefiting from our approximate query synthesis, PINOLO discovers 41 logical bugs in four mature DBMSs. At the time of the submission, 39 bugs have been confirmed by the developers. We hope that the promising results will put forward the study of DBMS testing, further promoting the reliability of database-backed systems.

Acknowledgments

We thank anonymous reviewers and the shepherd for their insightful comments. This work is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001), Natural Science Foundation of China (62272400), and Xiamen Youth Innovation Fund (3502Z20206036).

References

[1] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. Switchv: Automated sdn switch validation with p4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*, page 365–379, New York, NY, USA, 2022. ACM.

[2] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, Shing-Chi Cheung, and Haiming Chen. Semmt: A semantic-based testing approach for machine translation systems. *ACM Trans. Softw. Eng. Methodol.*, 31(2):1–36, 2022.

[3] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET workshop on Data description, access and control*, pages 249–264, 1974.

[4] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. *CoRR*, abs/2002.12543, 2020.

[5] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, 2018.

[6] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering*, pages 1257–1268. IEEE / ACM, 2019.

[7] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[8] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *Proceedings of 24th USENIX Security Symposium*, pages 193–206. USENIX Association, 2015.

[9] MariaDB Foundation. MariaDB Database. https://mariadb.org/, 2022. [Online; accessed Dec-2022].

[10] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. Griffin: Grammar-free dbms fuzzing. In *The 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[11] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 206–215. ACM, 2008.

[12] William E Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978.

[13] Free Software Foundation Inc. BNU Bison. https://www.gnu.org/software/bison/, 2022. [Online; accessed Jan-2023].

[14] Google Inc. american fuzzy lop - a security-oriented fuzzer. https://github.com/google/AFL, 2022. [Online; accessed Dec-2022].

[15] Google Inc. Go 1.20 Release Notes. https://tip.golang.org/doc/go1.20, 2022. [Online; accessed Jan-2023].

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226. ACM, 2014.

[17] MaraiaDB. Bug #30249 of MaraiaDB. https://jira.mariadb.org/browse/MDEV-30249, 2022. [Online; accessed Jan-2023].

[18] William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10(1):100–107, 1998.

[19] Michaël Mera. Mining constraints for grammar fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–418. ACM, 2019.

[20] MySQL. Bug #108937 of MySQL. https://bugs.mysql.com/bug.php?id=108937, 2022. [Online; accessed Jan-2023].

[21] MySQL. Bug #109407 of MySQL. https://bugs.mysql.com/bug.php?id=109407, 2022. [Online; accessed Jan-2023].

[22] MySQL. MySQL Database. https://www.mysql.com/, 2022. [Online; accessed Dec-2022].

[23] MySQL. The MySQL Test Framework. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN.html, 2022. [Online; accessed Dec-2022].

[24] OceanBase. A deployer and package manager for OceanBase. https://github.com/oceanbase/obdeploy, 2022. [Online; accessed Jan-2023].

[25] OceanBase. Issue #1100 of OceanBase. https://github.com/oceanbase/oceanbase/issues/1100, 2022. [Online; accessed Jan-2023].

[26] OceanBase. OceanBase Database. https://www.oceanbase.com/, 2022. [Online; accessed Dec-2022].

[27] PingCap. go randgen. https://github.com/pingcap/go-randgen, 2022. [Online; accessed Dec-2022].

[28] PingCap. Parser - A MySQL Compatible SQL Parser. https://github.com/pingcap/tidb/tree/master/parser, 2022. [Online; accessed Jan-2023].

[29] PingCAP. TiDB Database. https://github.com/pingcap/tidb, 2022. [Online; accessed Dec-2022].

[30] GNU Project. gcov——a Test Coverage Program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2022. [Online; accessed Jan-2023].

[31] qaqcatz. The Complete List of Instantiated Mutators in PINOLO. https://github.com/qaqcatz/impomysql_binary/blob/main/mutators.md, 2022. [Online; accessed Jan-2023].

[32] Mian Qin, Qing Zheng, Jason Lee, Bradley Settlemyer, Fei Wen, Narasimha Reddy, and Paul Gratz. Kvrangedb: Range queries for a hash-based key-value device. *ACM Trans. Storage*, jan 2023.

[33] Manuel Rigger. sqlancer. https://github.com/sqlancer/sqlancer, 2022. [Online; accessed Jan-2023].

[34] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1140–1152. ACM, 2020.

[35] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA):211:1–211:30, 2020.

[36] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 667–682. USENIX Association, 2020.

[37] Manuel Rigger and Zhendong Su. Intramorphic testing: A new approach to the test oracle problem. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 128–136. ACM, 2022.

[38] Andreas Seltenreich. SQLsmith: A random SQL query generator. https://github.com/anse1/sqlsmith, 2022. [Online; accessed Dec-2022].

[39] Donald R Slutz. Massive stochastic testing of sql. In *VLDB*, volume 98, pages 618–622. Citeseer, 1998.

[40] Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. Data-oriented differential testing of object-relational mapping systems. In *43rd IEEE/ACM International Conference on Software Engineering*, pages 1535–1547. IEEE, 2021.

[41] SQLite. How SQLite is Tested. https://www.sqlite.org/testing.html, 2022. [Online; accessed Dec-2022].

[42] TiDB. Issue #40015 of TiDB. https://github.com/pingcap/tidb/issues/40015, 2022. [Online; accessed Jan-2023].

[43] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Quadrant: A cloud-deployable NF virtualization platform. In *Proceedings of the 13th Symposium on Cloud Computing*, page 493–509, New York, NY, USA, 2022. ACM.

[44] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos Augusto M. Vieira, Ramesh Govindan, and Barath Raghavan. Galleon: Reshaping the square peg of NFV. *CoRR*, abs/2101.06466, 2021.

[45] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 724–735. IEEE / ACM, 2019.

[46] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, pages 328–337. IEEE, 2021.

[47] Shuai Wang and Zhendong Su. Metamorphic object insertion for testing object detection systems. In *35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1053–1065. IEEE, 2020.

[48] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 718–730. ACM, 2020.

[49] Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. Unicorn: detect runtime errors in time-series databases with hybrid input synthesis. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 251–262. ACM, 2022.

[50] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(1):15:1–15:28, 2022.

[51] Haoran Xu, Shuhui Fan, Yongjun Wang, Zhijian Huang, Hongzuo Xu, and Peidai Xie. Tree2tree structural language modeling for compiler fuzzing. In *Algorithms and Architectures for Parallel Processing - 20th International Conference*, pages 563–578. Springer, 2020.

[52] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 488–498. IEEE / ACM, 2019.

[53] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing SMT solvers via two-dimensional input space exploration. In *Proceedings of 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 322–335. ACM, 2021.

[54] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Skeletal approximation enumeration for SMT solver testing. In *Proceedings of 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1141–1153. ACM, 2021.

[55] Yuanyuan Yuan, Shuai Wang, Mingyue Jiang, and Tsong Yueh Chen. Perception matters: Detecting perception failures of VQA models using metamorphic testing. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16908–16917. Computer Vision Foundation / IEEE, 2021.

[56] Andreas Zeller. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.

[57] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970. ACM, 2020.

AutoARTS: Taxonomy, Insights and Tools for Root Cause Labelling of Incidents in Microsoft Azure

Pradeep Dogga
UCLA[*]

Chetan Bansal
Microsoft

Richie Costleigh
Microsoft

Gopinath Jayagopal
Microsoft

Suman Nath
Microsoft

Xuchao Zhang[†]
Microsoft

Abstract

Labelling incident postmortems with the root causes is essential for aggregate analysis, which can reveal common problem areas, trends, patterns, and risks that may cause future incidents. A common practice is to manually label postmortems with a single root cause based on an ad hoc taxonomy of root cause tags. However, this manual process is error-prone, a single root cause is inadequate to capture all contributing factors behind an incident, and ad hoc taxonomies do not reflect the diverse categories of root causes.

In this paper, we address this problem with a three-pronged approach. First, we conduct an extensive multi-year analysis of over 2000 incidents from more than 450 services in Microsoft Azure to understand *all* the factors that contributed to the incidents. Second, based on the empirical study, we propose a novel hierarchical and comprehensive taxonomy of potential contributing factors for production incidents. Lastly, we develop an automated tool that can assist humans in the labelling process. We present empirical evaluation and a user study that show the effectiveness of our approach. To the best of our knowledge, this is the largest and most comprehensive study of production incident postmortem reports yet. We also make our taxonomy publicly available.

1 Introduction

Cloud platforms and services, despite the best efforts of cloud providers, still suffer from production incidents and outages [13,15]. To improve reliability, cloud providers must first discover and resolve existing reliability risks [14]. Aggregating root causes of past incidents based on their post-incident reports (PIRs) is one effective approach to uncover common problem areas, trends, patterns, and risks (e.g., the most common root causes in the last year). Likewise, bucketing past incidents by their root causes can help on-call engineers (OCEs) to quickly retrieve and learn common mitigation strategies for a given root cause category. Such tasks are crucial for large-scale cloud platforms like Microsoft Azure that continuously strive to improve reliability by learning from past incidents caused by diverse factors.

PIRs are commonly written in natural language, with little structure. This makes the tasks of aggregating or bucketing past reports challenging, especially at a large-scale. One common practice to address this is to label each PIR with a *root cause tag* representing the category of its root cause. For example, a PIR for an incident caused by a code bug in the network driver can be tagged as "*Network.Driver.Code.Bug*". This enables quick and accurate aggregation of and retrieval from a large collection of PIRs simply based on their root cause tags instead of expensive and potentially inaccurate natural language processing of the PIR contents. For example, the tags can enable quickly answering questions such as: "how frequently did network driver code bugs cause incidents in the past year?" and "how were such bugs mitigated?"

Challenges and limitations. There are two key challenges in effectively labelling PIRs with root cause tags (more in §2).

The first challenge is to decide *what root cause tags to use to label the PIRs*. A well-defined taxonomy of root cause tags is essential for labelling PIRs consistently, otherwise different teams may tag the same root cause differently, hindering the cross-team aggregation analysis. A well-designed taxonomy for a large-scale cloud system such as Microsoft Azure should balance two competing objectives: it should be *comprehensive* enough to cover the myriad of potential root causes, yet *compact* enough for the OCEs to navigate and use easily. Moreover, it should be *fine-grained* enough to surface actionable insights from across many services.

Designing such a taxonomy is nontrivial. Several recent works analyzed production incidents and proposed taxonomies to capture their root causes. However, most of these works focus on specific root cause categories, such as software bugs [4,5,11,19,21,40], and thus their taxonomies are not comprehensive enough to cover other types of failures (e.g., hardware failures). Some other works consider multiple types of root causes, but they target specific services or

[*]Work done during internship at Microsoft

[†]Except for the first author, all authors are in order by their last names.

systems, such as big-data systems [38], a business data processing platform [9], or a particular cloud service [13, 14], rather than a large-scale cloud system. Moreover, existing taxonomies are not fine-grained enough to represent all the root causes we observe in Azure incidents.

Prior to our work, individual service teams in Microsoft designed their own root cause taxonomies based on domain knowledge. However, due to the lack of a comprehensive root cause labelling, many potential root causes were not anticipated when the taxonomies were designed. Consequently, a large portion of PIRs, whose root causes were not covered by existing taxonomies, were labelled with "Other" instead of more informative root cause tags (see §2 for more details).

The second challenge concerns *how to select root cause tags for incidents*. Currently, this is done manually—an individual (OCE) reads *lengthy* incident and post-incident reports, identifies the root cause, and chooses a suitable tag from a taxonomy that is often a long flat list of tags. This manual process is error-prone and can result in inconsistent tags across incidents—at Microsoft, tens of thousands of OCEs with varying levels of expertise and different interpretations of root cause tags conduct root cause analysis. We manually examined a small sample of 1241 PIRs and found that 29% of the OCE-assigned tags are incorrect. This problem might be mitigated if root cause tagging is done by a small group of experts through a stringent procedure, but this is infeasible at large scale systems like Microsoft Azure.

Contributions. In this work, we make the following three contributions that address the challenges described above.

First, we manually analyze over 2000 high-impact production incidents from 468 services in Microsoft Azure to identify a comprehensive set of root cause categories behind incidents in a large-scale cloud system. We carefully read the incident and post-incident reports and, if needed, consult the engineers from the affected service teams. Unlike previous empirical analyses of production incidents [4,5,9–11,14,21,38,40], we aim to identify not only a single root cause, but *all* factors contributing to the incidents. This analysis took more than four person-years and identifies 346 distinct root cause categories spanning all aspects of a production service, such as hardware and software, infrastructure and application, code and configuration, and so on. To the best of our knowledge, this is the most comprehensive empirical analysis of production incidents in cloud systems, considering the scale of incidents and affected services, the depth of analysis, and the diversity of root causes.

Our empirical analysis (§3) reveals several novel and interesting findings. For example, we show that most production incidents in real-world cloud systems involve multiple contributing factors; hence, preventing such incidents does not always require addressing all the causal factors, but only one (or a small subset) of them. This contrasts with existing research that focuses on a single root cause of an incident [10, 13, 21]. This finding implies that tagging a PIR with a single root cause does not capture the full picture of what caused the incident. Our analysis also shows that incidents result from many diverse factors spanning hardware and software, infrastructure and application, code and configuration, and so on. This, again, contrasts with existing research that focuses on a limited set of factors [4, 5, 9–11, 14, 19, 21, 38, 40]. We also show that the set of root causes evolves over time: new root causes emerge as new services or hardware are deployed, suggesting the need for a continuous root cause labelling effort. While our findings stem from analysis of incidents at Microsoft Azure, we believe that they generalize to similar large-scale systems and impact root cause analysis procedures at large.

Second, we propose a comprehensive taxonomy, the Azure Reliability Tagging System (ARTS), that organizes the root cause categories derived from our analysis. For ease-of-use, our taxonomy is organized hierarchically, with each leaf node representing a *root cause tag* describing a factor contributing to an incident. We expect that the root causes generalize to other cloud services and we open-source our taxonomy for the use of other researchers and practitioners.[1]

Finally, to reduce manual errors and inconsistencies in tagging PIRs, we developed AutoARTS, a tool that leverages machine-learning-based algorithms to assist humans. AutoARTS performs two key tasks: (1) it applies a multi-label classification technique to automatically analyze a PIR (written in natural language) and to extract multiple contributing factors and their corresponding tags from our proposed taxonomy; and (2) it generates a concise text snippet (from the PIR) that summarizes the relevant context for the factors. The snippet allows a human to quickly review the suggested tags without reading lengthy incident reports or PIRs. We describe how we adapt existing ML techniques for this purpose. Our empirical evaluation with real PIRs and a user study demonstrates the effectiveness of our approach. Specifically, our multi-label classification achieves an F1 score of 0.89. In the user study, our text snippets (contexts) received a score of 4.6 out of 5 (5 being 'very useful'), contained *no* unnecessary details, and helped an expert identify two additional contributing factors (in a set of ten incidents) that they had originally missed.

Deployment status. Most of the 468 services whose production incidents we analyze have been deployed as part of Microsoft Azure for several years. High-impact incidents in these services have been analyzed and labelled with ARTS tags since November 2020. The labels, available to all services in Azure, are regularly aggregated to identify key problem areas and to devise actionable insights (examples in §4).

2 Background and motivation

2.1 Background

Incident Reports. Incidents are unplanned outages that impact production services and their users. The severity of an in-

[1] ARTS Taxonomy: https://autoarts-rca-taxonomy.github.io

cident may vary based on aspects such as the criticality of the affected services and the number of impacted users. As described in [31], an incident's life-cycle is a complex process involving several steps such as detection, triaging, diagnosis, root cause analysis, mitigation, and resolution. An *incident report* documents important information related to these various steps. At Microsoft, an incident report can be created by impacted users as well as by automated monitors and it usually contains the following: (1) a concise title, (2) a summary of the incident, highlighting some events in the timeline from detection to mitigation, (3) engineers' discussion thread to share relevant information corresponding to the incident's resolution, and (4) several other fields such as severity, owning team, time to mitigation, mitigation steps, etc.

Post-Incident Reports (PIR). After an incident is resolved, a best practice is to conduct a retrospective or postmortem analysis of the incident to produce a *post-incident report* (PIR). In a PIR, the service team reflects on what went wrong, why it went wrong, what they learned from it, and how to avoid similar incidents in the future. At Microsoft, a PIR is a natural language document and it contains sections such as (1) root cause summary that describes all root causes, (2) five-whys [34] that iteratively drills-down the cause-and-effect relationships of various contributing factors, (3) preventive measures for similar future incidents, and so on. Generating a PIR requires significant effort, and hence, only the incidents with high severity are required to have them at Microsoft.

Root cause labelling. A critical component of a PIR is its *root cause tag* that represents the root cause categories of the incident as determined by a postmortem analysis. For example, a root cause tag *"Authoring.Code.Bug.RaceCondition"* indicates that the incident is caused by a race condition in software code. Such concise labelling allows one to efficiently and accurately aggregate and summarize a large number of historical PIRs solely based on their tags, without expensive and potentially error-prone natural language processing of the PIR contents. These aggregate results are regularly reviewed by the engineering leadership to find global trends (e.g., frequent root cause categories), which guide business decisions such as prioritizing engineering investments. These tags also simplify information retrieval and knowledge sharing: an engineer seeking to actively mitigate an ongoing incident caused by a network driver can quickly retrieve and consult past PIRs with root cause tag *Network.Driver*. For effectiveness of aggregation and retrieval, it is important that the root cause analysis and labelling process is accurate.

At Microsoft, authors of incident reports or PIRs can label their reports with root cause tags selected from taxonomies of root cause categories that are predefined by domain experts.

2.2 Challenges in root cause labelling

We analyze a sample of $\approx 1.7M$ root cause analyses in Microsoft, across all its services, to understand the challenges in root cause labelling. We now summarize the key findings.

Finding 1. *Existing taxonomies, although designed by domain experts, are not comprehensive enough.*

This is due to the lack of a comprehensive study of root causes, many potential root cause categories are missed or not anticipated when a taxonomy is designed. As an implication, a PIR author may not find a suitable predefined root cause tag to describe the current incident. In our sample of root cause analyses, $\approx 20\%$ incidents are labelled as 'Other' and $\approx 58\%$ are labelled with categories containing 'Other' (e.g., 'Network - Other'), implying that their root causes are not covered or only partially covered by the existing taxonomies. Such 'Other' tags are not useful in the aforementioned root cause aggregation and retrieval tasks.

Finding 2. *Existing manual root cause labelling process is expensive and error-prone.*

Root cause label of an incident is often determined based on its PIR and incident report. These documents are usually *long* (4542 words per incident in our sample) and *complex* (on average, ≈ 9 engineers engaged in discussion exchanging 20 comments). Thoroughly understanding these long documents to identify all contributing factors behind an incident, and then selecting from predefined root cause labels that represent the factors, is a nontrivial task.

Even when the root cause is understood, PIR authors may make mistakes in choosing the correct tag. This can happen due to multiple factors. Existing taxonomies at Microsoft are flat long lists, making it difficult to navigate through them and to pick the right tags. Moreover, many individuals are involved in the rootcausing efforts. For example, we observe $34K$ distinct individuals involved in a sample of $600K$ PIRs in Microsoft. This large number of individuals are likely to have varying degrees of expertise and different interpretations of root cause tags. This is further exacerbated by ambiguous or confusing tags in the taxonomy (e.g., 'Network' and 'Datacenter - Network'). All these factors can contribute to inconsistent and/or inaccurate labels. We manually examined a small sample of 1241 PIRs and found that 29% of the assigned tags are incorrect.

Our goals. In this paper we address the challenges above with a three-pronged approach. First, to address the first challenge above, we conduct an extensive multi-year analysis to identify a wide variety of fine-grained root causes of 2000+ production incidents from across 450+ services in Microsoft Azure. Second, based on this analysis, we propose a novel hierarchical and comprehensive taxonomy of potential contributing factors behind the incidents (§4). Third, to address the second challenge above, we develop an automated tool that can assist a human by presenting necessary context from PIRs that identify contributing factors to reduce cognitive load and improving accuracy in the root cause labelling process by suggesting tags from the taxonomy (§5).

3 Empirical Analysis of Production Incidents

3.1 Goals and Methodology

There exists several empirical studies of production incidents in large-scale cloud systems [4, 5, 9–11, 14, 21, 38, 40]. We have two goals that differentiate our study from them. First, we do not restrict our analysis to a limited set of root cause categories (e.g., software bugs [4, 5, 11, 19, 21, 40]) or specific services/platforms (e.g., big-data systems [38]). Second, for each incident, we try to identify not a single root cause, but *all* factors contributing to the incidents. These two goals enable us to identify a wide-range of contributing factors behind incidents happening in large number of services/platforms.

We analyze 2051 high-impact incidents in 468 Microsoft Azure services. We carefully analyze each incident by carefully reading and understanding its incident report and PIR, the discussion comments, and even the work items (e.g., bug fix, system upgrade) that are created due to the incident. When something is not clear, we reach out to the incident owners to clarify. As a part of the analysis, we not only identify the *contributing factors* causing the incident but also extract text snippets or *context* from the incident and PIR which helps explain and justify the identified root cause tag for future reference and validation. Every week, we peer review a randomly selected subset of incidents to help us refine our collective understanding of tag usage, promote learning and improve accuracy. If we identify a new category of root causes, which is not covered by existing tags, we then propose new tags which are internally reviewed before getting introduced to the taxonomy. For any tag in the taxonomy, we also provide it's description in natural language for future reference. This data lives in an internal database which can be easily joined with incident databases and visualization reports are created for easy data analysis based on various pivots such as contributing factors, services, incident impact, etc. We also meet with the engineering teams on a weekly basis, and review our data both for accuracy and to share insights that result in reliability improvements.

Our root cause analysis effort is guided by several principles: (1) Our analysis is intellectually honest so that individual teams that conduct their own postmortem analysis (to identify a single root cause) feel psychologically safe, valued, and included; (2) Our analyses have meaningful depth because we start by capturing customer pain and go down to "work as done" by our engineers; (3) We focus on both depth and breadth of incident analysis enabling us to highlight thematic learning that broadly improves Azure services and the underlying cloud platform; (4) Our findings are turned into new standards or updates to existing standards and are actionable and useful because they address customer pain; and (5) While learning is important; continuous learning is necessary and crucial.

The above process is required to ensure high quality of root cause analysis. However, it needs significant manual effort. Since 2020, we have analyzed 2051 incidents in 468 Azure services with a team of 2-4 members.

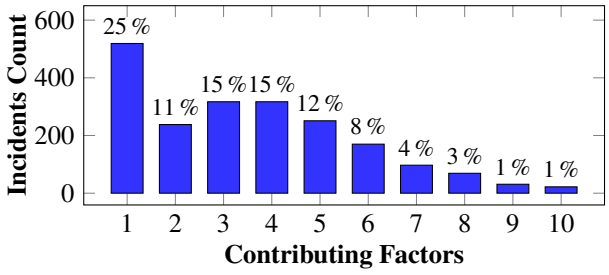


Figure 1: **Distribution of incidents across number of distinct contributing factors (shown until 10 factors).**

3.2 Analysis Results

We here present several interesting findings from our analysis.

Finding 3. *Incidents are often caused by multiple contributing factors working together instead of an isolated root cause.*

This is contrary to prior work [13, 15, 16, 21] that focus on identifying a single root cause per incident. Consider, for example, a real incident where a service became unavailable after a single customer continuously pushed a load that was 60x greater than what the service was scaled to handle. The original PIR author chose the root cause label "*Service – Load Threshold*." This itself is not an inaccurate root cause when forced to pick only one cause. However, there are many more factors involved in this incident: (1) there was an inrush of load from a single customer, (2) there was a lack of throttling on the customer end as well as the service end, (3) increased load significantly increased CPU and heap usage and thread count at the server, which lead to failed requests with exceptions, (4) the exception handling didn't release some resources that were allocated by the failed requests, leading to resource leaks, (5) there were no automated watchdogs to detect early symptoms of the outage (or resource leaks), and (6) the team was unable to access their own metrics during the outage since the metrics were collocated with the service. In contrast, our analysis of the incident identifies all these factors and the corresponding tags in our taxonomy.

Figure 1 shows the distribution of the number of contributing factors behind each incident. As shown, over 75% of incidents have been caused by more than one contributing factors. And, more than 50% of the incidents have 4 or more contributing factors. On average, each incident has ≈ 3.6 factors. This reaffirms the need for holistically analyzing the incidents to understand all the contributing factors.

The presence of multiple contributing factors per incident has important implications. On one hand, identifying the possibility of such incidents before deployment to production with integration and end-to-end tests is challenging since test-

Category	Description	Frequency	TTM (Hrs)
Detection	Issues related to detecting problems before they affect production	**61%**	50
Authoring	Issues in authoring artifacts like code, config, troubleshooting guides, etc.	**50%**	**58**
Dependency	Issues in a dependency the service has, most typically another service but can also be some things where a boundary between teams is present	**37%**	16
Architecture	Issues in how the service is architected and likely where any work to fix would require changes to the architecture of the service	20%	33
Deployment	Issues related to deployment of code or config	20%	27
Process	Any issue caused by human errors, a flawed process or the lack of a process	18%	**123**
Load	Any issue caused by the service not being able to handle changes in load	14%	13
Auth	An authentication or authorization related issue	7%	21
Performance	An issue that caused excess latency	6%	16
Datacenter	Events (hardware, installations, power interruption, etc.) in the datacenter	4%	**70**

Table 1: **High-level root cause categories from ARTS taxonomy with their descriptions, frequency of occurrence in our analysis and mean Time-To-Mitigate (TTM) for incidents caused by their sub-categories.**

ing needs to be performed in the presence of multiple potential contributing factors (e.g., high load *and* no throttling *and* no monitoring of early symptoms). On the other hand, *preventing such an incident does not always require addressing all the causal factors, but only one (or a small subset) of them.* For example, the aforementioned incident could have been prevented by using proper throttling mechanism, *or* by fixing the resource leak bug, *or* by having monitors that can restart the service on early symptoms of resource leaks. This insight presents a unique opportunity to fix the incidents (by addressing the easiest causal factor); but it requires identification of all the causal factors (as we do) instead of identifying a single root cause.

Finding 4. *A wide-variety of factors contribute to production incidents.*

Our analysis identified a wide range of factors, including hardware, software, code bugs, underlying infrastructure to external dependency issues, configuration errors, deployment issues, and so on. Specifically, we have identified 346 root cause categories (i.e., contributing factors) for the 2051 incidents we analyzed. Table 1 shows the high-level root cause categories, each of which contains many finer-grained sub-categories. The full list of categories and their respective frequencies observed in our analysis can be found in https://autoarts-rca-taxonomy.github.io. This contrasts our study with prior works that focus on a small set of root causes such as code bugs [4, 5, 11, 19, 21, 40].

Finding 5. *New root-cause categories keep appearing over time.*

As software and hardware systems evolve, novel root causes appear to contribute to their incidents. For example, when a service migrates to a containerized environment, its incidents may be caused by container-related factors. Similarly, when a service takes a new external dependency, it may start experiencing incidents caused by factors related to the failures of the new dependency. We analyze incidents in the same timeline as they appear and we create root cause categories incrementally—we create a new category only if none of the existing ones can precisely represent a new root cause. We observe that even though many common root cause categories (e.g., code bugs) appear in early incidents that we analyze, a few categories appear only in much later incidents (e.g., those happening two years after the first incident we analyzed). Figure 2 shows how often such new categories appear in our analysis. As shown, even after 1.5 years, new root cause categories appear, albeit with a smaller rate (i.e., we can tag higher number of incidents successfully before we need to introduce a new root cause category). The fact that novel root cause categories keep appearing implies that root cause labelling needs to be a continuous process to identify (and take actions on) emerging root cause categories. This calls for an automated solution.

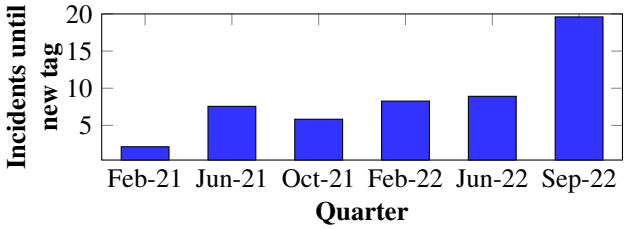


Figure 2: **Average number of incidents successfully tagged until a new root cause tag is introduced across quarters.**

Finding 6. *Lack of monitoring (i.e., observability) is the most common factor behind incidents.*

Table 1 shows the distribution of various contributing factors behind the incidents we analyzed (only high-level factors are shown). As shown, Detection is the most common contributing factor leading to outages. Detection related issues represent missing observability signals that prevent us from detecting early symptoms of problems, many of which could have been avoided, e.g., by rebooting the service, if their early

Contributing Factor	Frequency
Detection.Monitoring.MissingAlert	34%
Authoring.Code.Bug.Change	25%
Detection.Monitoring.InsufficientTelemetry	18%
Detection.Validation.MissingTestCoverage	17%
Detection.Monitoring.CodeDeployment. InsufficientHealthSignal	9%
Authoring.Documentation. NoOrInsufficientTSG	9%
Architecture.SinglePointOfFailure	8%
Authoring.Code.Bug.Latent	7%
Detection.Monitoring.Synthetic	6%
Deployment.Mitigation.ManualTouch	5%

Table 2: **Distribution of top 10 most frequent contributing factors in our analysis from the ARTS taxonomy.**

symptoms were detected. We also analyzed finer-grained contributing factors from ARTS taxonomy. Table 2 shows distributions of the top ten contributing factors (a contributing factor X.Y.Z means factor Z is a specific case of factor Y, which is a specific case of factor X). As shown, Missing Alerts, which is a specific case of Monitoring, which is a specific case of Detection, is the most common contributing factor. Insufficient telemetry captured from services is also a major contributing factor which also prevents from deploying automated alerts. An organizational policy on collecting key telemetry and defining automated watchdogs informed by this aggregate analysis can mitigate incidents (or severity) in the future.

We also analyze the most frequently *co-occurring* root causes to identify the pairs that jointly cause incidents. The two most frequent pairs are "*Authoring.Code.Bug.Change*" & "*Detection.Monitoring.MissingAlert*" (15%) and "*Authoring.Code.Bug.Change*" & "*Detection.Validation.MissingTestCoverage*" (11%). This aligns with our experience that many production incidents are caused by buggy code changes that are deployed without proper monitoring and testing.

Finding 7. *Incidents caused by deployment and datacenter related issues are the most time consuming to mitigate.*

In incident management, TTM is defined as the time elapsed between the start of the incident and when its customer impact was completely resolved. The higher the TTM, the more the customer impact and dissatisfaction. From Table 1, we can see that incidents caused by Process and Datacenter related root causes have the highest mean TTM. Process related incidents have a high TTM because these incidents are caused by human errors and lack of standard operating procedures which result in non-trivial hard-to-resolve issues (e.g., accidental deletion of a database). Datacenter related incidents are caused primarily due to hardware failures which are quite complex given that there are multiple layers of capacity buffers all of which need to fail before an incident is caused by hardware issues.

4 Root Cause Taxonomy

We organize the root cause categories identified in our empirical study as a taxonomy of reliability tags that can be used to label PIRs of incidents.

Design goals. We have the following design goals in designing the taxonomy. First, the taxonomy should be *comprehensive* enough to capture not only the primary root causes of past incidents in Azure, but also other (secondary) contributing factors. Second, in order to avoid having a taxonomy too large to be easily used in practice, the taxonomy should be *sufficient* and it should include only the root causes found in past incidents. This implies that the taxonomy is continuously and organically grown to include new categories as they are discovered. Third, the tags should be *unambiguous*, to enable high-quality annotations. Finally, the taxonomy is organized *hierarchically*, for ease of labelling and updates.

The ARTS taxonomy. We achieve the goals with a novel taxonomy called ARTS (Azure Reliability Tagging System) taxonomy. The taxonomy is built on top of the root cause categories identified by our empirical analysis described before. We start with a small number of tags representing orthogonal categories of themes (such as datacenter issues and authentication issues) and grow it horizontally to include new themes and vertically to include more specific sub-themes as new incidents are analyzed and existing themes/sub-themes deem inadequate. We have established a continuous feedback loop based process for building the ARTS taxonomy and tagging of new incidents on an ongoing basis.

For ease-of-use, we organize the ARTS taxonomy hierarchically, by grouping related sub-themes under one common theme. Currently it consists of four levels and contains 346 root cause categories identified from our empirical analysis. The top level consists of ten broad themes (shown in Table 1), each of which consists of multiple sub-themes. There are 346 leaf nodes, each representing a root cause tag with the name obtained by concatenating the names of the path from the root to the leaf node. For example, the root cause of "a gap in preproduction detection due to missing integration tests" is represented with the tag "*Detection.IntegrationTest.Missing*" in which "*Missing*" is the most precise leaf-level tag. The hierarchical taxonomy naturally distinguishes between problem spaces at different granularities. In this example, if the root cause is that the integration tests existed but were skipped somehow, that representative tag would have the leaf-level tag "*NotRun*" instead of "*Missing*".

As mentioned, the taxonomy is grown as new root causes are identified in newly analyzed incidents. Figure 2 shows how the taxonomy has been growing over time, with the y-axis showing the average number of incidents analyzed until a new tag needed to be introduced in ARTS. A larger value indicates better stability of the taxonomy: many incidents can be analyzed with existing tags. As shown, over time, the taxonomy can be seen becoming stable. Specifically, in the

most recent quarter, only one new tag needed to be introduced after analyzing ≈ 20 incidents (i.e., after using ≈ 70 existing tags) on average. We hope to see significantly more stability in coming months.

For lack of space, we omit further discussion about 346 leaf nodes in the taxonomy. However, we open source the taxonomy, with all tags and their descriptions, at https://autoarts-rca-taxonomy.github.io. We believe our open-source effort will foster future research and allow practitioners use our taxonomy. Even though the ARTS taxonomy is developed based on incidents in Azure, we believe that its categories are general enough to be used in any large-scale cloud system.

Deployment Status. The ARTS taxonomy and PIRs labelled with ARTS tags have been available to Microsoft engineers since when we started building it (November 2020). It enables various service teams within Microsoft Azure to systematically look at past incidents. The learnings have been valuable, especially for service teams without the required expertise or resources to dig deep into their service reliability. They have enabled engineering teams prioritize work items and often times this also results in creation of new engineering standards and tools. For example, lack of unit testing was a common factor contributing to many incidents in a large organization in Azure—incidents were caused by bugs in code that was not tested (or poorly tested). This caused the organization to enforce the policy that all checked-in code must have sufficient tests to achieve a certain code coverage. Lack of throttling was another factor highlighted by aggregating ARTS labels; this started a new engineering group with the goal of building a common throttling service that all Azure services can easily use. Last but not the least, engineering teams emphasize on using more extensive monitoring/observability tools, since as ARTS labelling showed, adequate monitoring could prevent many incidents.

Overall, Microsoft engineers found the ARTS labels in their PIRs useful. Here we show samples of the verbatim feedback from the service owners at Microsoft: *"Tracking incidents against a known set of root causes is extremely useful. Your effort has enabled us to make data-driven decisions, and already produced several benefits in a short time." "Your data established clearly that services that have high maturity in certificate management had fewer outages. This validated that our investment across Azure is in the right direction." "The ARTS report is an important resource for us for data driven planning related to Azure Quality."*

5 AutoARTS for Automated Labelling

As mentioned before, identifying and labelling PIRs with their root causes is expensive and error-prone. To reduce the cost and errors, we have developed an automated tool called AutoARTS that can assist a human in the labelling process with two important tasks. First, it uses a multi-label classification technique to automatically analyze an incident's PIR (written in natural language) and to identify multiple contributing factors and their representative ARTS tags. Second, it can produce a short text snippet (from the PIR) that captures important context explaining the factors. The snippet enables a human to easily review the selected tags without reading lengthy incident reports or PIRs. We now describe how AutoARTS performs these two tasks by using ML techniques. Figure 3 shows the architecture and components of AutoARTS.

5.1 Automatic identification of ARTS tags

AutoARTS uses multi-label text classification to classify a PIR into a set of ARTS tags. One key challenge we face is that conventional multi-label text classification algorithms that treat each class as opaque and independent, require sufficient labelled data for each class to achieve good accuracy. However, even though we have a reasonable collection of labelled data, many of the fine-grained classes (i.e., ARTS contributing factors) contain very few labelled samples (i.e., PIRs). Specifically, 68% classes have fewer than 10 labelled samples in our dataset, which can adversely affect classification accuracy.

To address this, we leverage the hierarchical relationship between root cause labels by encoding the taxonomy structure using Graph Convolutional Networks [18]. Exploiting the structure of the taxonomy enables transfer of knowledge from the categories with adequate labels to categories with few labels (§6.2). In particular, we apply a hierarchical text classification model called HiAGM [42]. Contrary to the conventional multi-label text classification methods that disregards the holistic label structure for label correlation features, this model attempts to fully utilize the mutual interactions between the text feature space and label space, as well as label dependencies. As an illustration, consider the root cause taxonomy in Figure 3. The "*Authoring.Code.Change*" category only contains 13 samples, making training difficult owing to the small amount of labelled samples. However, by modeling the root cause taxonomy as a graph, we can transfer knowledge from "*Authoring.Code.Bug*", which has 733 labels, to "*Authoring.Code.Change*" since they share the features of their same parent root cause category, "*Authoring.Code*".

Given a Post-Incident report $x = (w_1, w_2, \ldots, w_n)$ with n tokens, the sequence of token embedding is initially fed into a bidirectional GRU neural network [6] to extract text contextual features. Following the GRU model, multiple CNN layers are employed to generate the n-gram features. The top-k max pooling layer is then applied to obtain the overall text representation $S \in \mathbb{R}^{n \times d_c}$ that highlights the key information, where n is the top-k output dimension of CNN layers and d_c represents the embedding dimension.

To model the ARTS taxonomy, we formulate the taxonomic hierarchy as a directed acyclic graph $G = \{V, E_t, E_b\}$, where V refers to the set of label nodes. E_t and E_b represent the top-

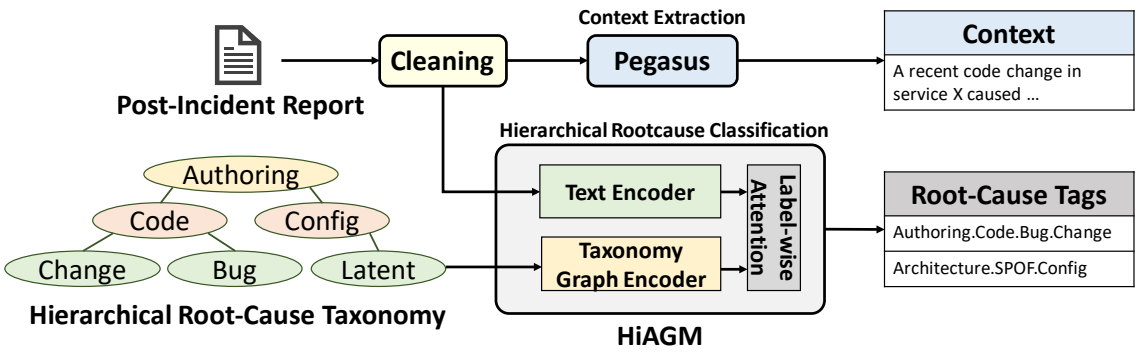


Figure 3: **Overview of our Context Extraction and Hierarchical root cause Classification using Post-Incident reports.**

down and bottom-up hierarchy paths respectively. To encode the hierarchy graph, a GCN-based hierarchy encoder [18], Hierarchy-GCN, is used to aggregate data flows within the top-down, bottom-up and self-loop edges based on the associated neighborhood of each node. The GCN-based graph encoder adapts the convolution concept from images to graphs, in which the graph convolutional operator can effectively convolve the multi-order neighborhood information by forming multiple propagation steps during the forward pass. For each node, the feature information is aggregated by the node feature from all the neighbors, including the node itself, to leverage the graph structure of label taxonomy.

Next, we aggregate the features of texts and labels together using a label-wise attention mechanism [37]. Specifically, the attention α_{kj}, which indicates how informative the j-th text feature is for the k-th label, is calculated as follows:

$$\alpha_{kj} = \frac{e^{\mathbf{s}_j \mathbf{h}_k^T}}{\sum_{j=1}^{n} e^{\mathbf{s}_j \mathbf{h}_k^T}}, \tag{1}$$

where $\mathbf{s}_j$ is the j-th text feature of the root cause input and $\mathbf{h}_k$ represents the k-th node in the label hierarchy. The label-aligned text feature $\mathbf{v}_k = \sum_{i=1}^{n} \alpha_{ki} \mathbf{s}_i$ for the k-th label is then obtained and fed into a classifier for hierarchical label prediction.

Finally, we flatten the label hierarchy by treating all nodes as leaf nodes for multi-label classification, regardless of whether a node is a leaf or an internal node. A binary cross-entropy loss function is employed to train the model using the ground truth and predicted sigmoid score for each label. In additional, a recursive regularization for the parameters of the final fully connected layer is used to encourage classes nearby in the hierarchy to share similar model parameters.

$$L_r = \sum_{i \in C} \sum_{j \in child(i)} \frac{1}{2} \|\mathbf{w}_i - \mathbf{w}_j\|^2, \tag{2}$$

where the node j is the child of node i in the label hierarchy.

5.2 Context Extraction from PIR

The objective of context extraction is to extract key text snippets from a given Post-Incident Report (PIR) for on-call engineers to comprehend the contributing factors of an incident without reading the complete lengthy report. Many text summarization techniques exist. *Abstractive summarization*, where summaries may contain generated sentences, is not a good fit for us since our goal is to select and highlight existing texts in the PIR, as is done by *extractive summarization*. However, existing language models such as BERT [8] and XLNet [36] are trained on large corpuses such as Wikipedia articles, etc., where the syntax and semantics of the language used is quite different from what is observed in PIRs due to domain-specific usage of words (e.g., 'Fabric' in networking terminology vs clothing) and different vocabulary. Moreover, existing extractive summarization models are trained on and their traditional usage in *summarizing* text documents, which is different from context extraction from PIRs. Our experimental evaluation in §6.3 shows that they perform poorly. We therefore finetune an existing model called Pegasus [39] with our labelled data (from our empirical study described before).

In Pegasus, important sentences are removed or masked from an input text and are generated together as one output sequence from the other remaining sentences, similar to an extractive summary. Hence, Pegasus is amenable for context extraction from PIRs, because we can mask the key sentences identified in our analysis (§3.1) to finetune the model parameters. Using the standard Transformer encoder-decoder, Pegasus model is pre-trained on two enormous text corpora: 1) Colossal and Cleaned version of Common Crawl (C4) [27], which comprises of text from 350 million web pages with a size of 750 gigabytes; 2) HugeNews [39], a dataset of 1.5 billion news articles gathered from 2013 to 2019. Similar to MLM tasks for predicting masked tokens, a new pre-training task called Gap Sentences Generation (GSG), is applied to fill the masked sentences. Three different strategies are applied for selecting m gap sentences without replacement from a document. The first method is to uniformly select m sentences at random, whereas the second strategy is to simply select the first m sentences. The aforementioned two strategies are combined with the Principal strategy, in which top-m scored sentences are chosen based on their significance as measured by rouge score [20] without the selected sentence. Formally, the

The root cause of this monitor alert was that a lot of subscriptions could not deploy VMs on indiacentral region.
... ... *(omit 132 words)*
This problem occurred because the traffic that was re routed to AZSM could not be handled. This problem occurred on indiacentral prod b and indiacentral prod b. As part of increasing inventory we have introduced news sets of AMD clusters. The AZSM services on these clusters still needed some configuration and build out related processed to be completed. Hence these clusters stamps could not handled the traffic re routed to them. The traffic was routed as part of default behavior. We are going to change this. The Fabricator clusters started taking tenant traffic even though their corresponding Az SM clusters weren't ready. This was done as part of flighting on Broad clusters. India Central was one of the region for this flighting. We did not anticipate a case where new build out clusters would not be able to take the new traffic. This was detected as part of the API failure monitor. We will be working on adding more robust feature specific monitoring and more strict rollout to not encounter this failure again.

Figure 4: **Context extraction from a redacted PIR. Green sentences are extracted by both our model and human expert, red are extracted by model only and blue are extracted by human only.**

score s_i of the i-th sentence x_i can be expressed as follows:

$$s_i = \text{rouge}\big(\text{x}_\text{i}, \text{D} \setminus \{\text{x}_\text{i}\}\big), \tag{3}$$

where the D is the set of all the sentences in the document and rouge function is a commonly employed metric for evaluating how good an automatically produced summary against a reference summary.

Even though Pegasus has been pre-trained on massive datasets, it is not trained to generate context from root cause descriptions in software engineering domain. To completely comprehend the context extraction task in our domain, we utilize the human-labeled context to further fine-tune the Pegasus model as a sequence-to-sequence task.

Based on the empirical results in §6.3, we found the Pegasus model can outperform the state-of-the-art abstractive summarization technique on our dataset for the reasons listed below. The human-labeled contexts are chosen straight from the original root cause details with no alteration to the phrase structure. Pegasus, an abstractive summarization model that pre-trained on the Gap Sentences Generation task, may immediately replicate the important sentences from the input, resulting in a higher overall rouge score than the traditional abstractive technique.

Figure 4 illustrates an example of context extraction from a PIR report consisting of 328 words. The human-labeled context is shown in blue and green, whereas the context extracted from Pegasus is shown in the green and red. This example shows that ≈50% tokens can be filtered, which can considerably enhance the efficiency with which on-call engineers read the PIR report. Also, we discover that the extracted context from Pegasus has a high recall compared to the human labels, allowing engineers to seldom overlook vital information.

Section	Micro-F1	Weighted-F1
Whole PIR	**0.55**	**0.40**
Title	0.53	0.45
Summary	0.47	0.46
RC-Details	0.52	0.45
5-Whys	0.54	0.40
Discussion	0.53	0.40
Mitigation	0.47	0.40
RC-Details + 5-Whys	**0.56**	**0.42**

Table 3: **Study on the utility of different PIR sections in top-level root cause classification using Random Forests.**

6 Evaluation

We now empirically evaluate the performance of AutoARTS.

Dataset. Our dataset consists of 1120 PIRs that are expert-annotated with ARTS root cause tags and contextual sentences to justify them. We use stratified sampling to divide this dataset into train(72%) and validation(8%) splits to train and tune the hyperparameters of different models and test(20%) split to report the results with the trained models.

Data Pre-processing. We found that engineers often included various types of data such as debugging queries issued on logs, error messages, stack traces, screenshots, etc., in PIRs (also identified in [31]). These add significant noise to the vocabulary of the language processed by NLP models, without contributing to performance. We carefully remove such noise with regular-expression based filters and only select alphabetic text for our evaluation. For experiments in §6.1, we also use the NLTK [3] library to remove stop-words and extract stems of words to construct vocabulary.

Evaluation Metrics. For root cause classification, we use micro-F1 score to analyze performance across different incidents with multiple labels. We also use weighted-F1 score to analyze performance across different classes since our dataset is imbalanced as shown in Table 1. For context extraction, we use ROUGE (Recall-Oriented Understudy of Gisting Evaluation) [20] and BLEU (Bi-Lingual Evaluation Understudy) [25] scores to evaluate the similarity of extracted context against the ground truth. Rouge-N score is based on the percentage (higher the better) of N-grams from the ground truth that are present in the extracted context. BLEU-N score indicates the percentage of N-grams from the extracted context that are present in the ground truth. Rouge-L F1-score is based on the longest common subsequence (not necessarily consecutive) between the extracted context and target context.

6.1 Featurization and Feature Selection

R1: Given the input text sequence length limitations of DL models, what information should be used from PIRs?

Sophisticated DL language models impose constraints on input sequence length (e.g., 512 tokens for BERT [8]). The

Model	ROUGE			BLEU			
	Rouge-1	Rouge-2	Rouge-L	BLEU	BLEU-1	BLEU-2	BLEU-3
Pegasus - Pretrained	32.55	18.72	24.30	9.61	18.03	10.31	8.93
Pegasus - Finetuned	**45.46**	**35.65**	**38.43**	**24.60**	**32.19**	**24.98**	**23.41**
T5 - Pretrained	34.38	23.31	28.03	10.06	15.68	10.83	9.43
T5 - Finetuned	41.63	33.86	35.76	23.81	29.81	24.10	22.70
BERT-cased - Pretrained	40.05	27.03	31.01	18.62	28.43	18.95	16.83
BERT-cased - Finetuned	40.08	27.35	31.20	18.80	28.32	19.03	16.95
BERT-uncased - Pretrained	39.52	26.58	30.74	17.63	27.47	17.98	15.89
BERT-uncased - Finetuned	39.92	27.44	31.57	18.64	28.08	18.91	16.90

Table 4: **Performance of Pegasus and T5 models with their corresponding pre-trained versions and fine-tuned versions. We also present performance of unsupervised clustering based approach for extractive summarization using BERT.**

Model	Micro-F1	Weighted-F1
HiAGM	83.16	89.63
HiAGM_Flat	45.40	68.66
BERT_MLC	42.29	46.85

Table 5: **Performance of HiAGM compared to using flattened root cause taxonomy (HiAGM_Flat) and a finetuned-BERT based multilabel classifier (BERT_MLC).**

Model	Test Perplexity	Test Accuracy
BERT-uncased	7.57	34.83%
BERT-cased	6.69	35.26%

Table 6: **Performance of MLM based pre-finetuning and OCE-assigned root cause based finetuning of BERT.**

limit is much smaller than our preprocessed PIRs (avg. length of $\approx$ 1900 words). However, a PIR is organized into multiple sections and we conduct an ablation study by featurizing each section in the PIR into Bag-of-Words encodings and classify them to top-level root cause categories using a Random Forest classifier. Table 3 highlights that "root cause details" and "5-Whys" sections achieve better micro-F1(1.8% higher) and weighted-F1 (5% higher) scores compared to using the whole PIR. These sections capture information relevant to root cause classification and by only using them, we minimize sequence length to meet constraints imposed by DL models.

6.2 Hierarchical Classification

R2: Is the hierarchical nature of our taxonomy beneficial in leveraging relationships between root causes to classify incidents?

Table 5 compares the level-3 root cause classification performance of our trained HiAGM model against a flattened version of our hierarchical taxonomy (HiAGM_Flat), where we remove parent-child relationships between different root causes in the taxonomy and consider all the root causes tags to be opaque and independent of each other. We also compare HiAGM against a multi-label classifier (BERT_MLC) with the flattened version of the taxonomy using finetuned BERT [8] model (details in §6.4) to encode the PIR text. We observe that HiAGM performs significantly better (31% higher weighted-F1 measure) than HiAGM_Flat indicating the utility of GCN to leverage neighboring relationships between root causes and the need for root cause taxonomies to be hierarchical. HiAGM performs significantly better (91% higher weighted-F1 measure) than BERT_MLC along with HiAGM_Flat (47% higher weighted-F1 measure), demonstrating no utility in finetuning existing language models on PIRs.

6.3 Context Extraction

R3: How do supervised (abstractive and extractive) or unsupervised (extractive) summarization approaches fare for context extraction? Is finetuning necessary for context extraction and does it work with limited data?

Using the train and validation splits of the dataset, we finetune Pegasus for 15 epochs and T5 (3 Billion parameters) [27] for 7 epochs and report the results on the test set. Table 4 compares the performance of finetuned Pegasus model against baseline approaches using T5 and clustering-based extractive summarization [23] using BERT. We can clearly see that our finetuned Pegasus model achieves the highest performance across various ROUGE and BLEU metrics. We observe a significant (58.15%) improvement in Rouge-L score as a result of finetuning Pegasus, because pre-trained version of Pegasus is trained on significantly different domains of language such as news articles, etc., and is trained to summarize them, which is different from context extraction. We also observe a 7.6% increase in ROUGE-L score compared to the finetuned-T5 model, because Pegasus extracts sequences of text from the PIR as opposed to T5 which generates new sequences of words, which might not represent the content present in the PIR which is where engineers derive their context from. Finetuned-Pegasus performs significantly higher (21.73%) than unsupervised clustering based summarization approaches using finetuned-BERT, because summary of the PIR doesn't represent the context.

Metric	Response	Description
(Q1) Usefulness of generated context to identify contributing factors	4.6/5	1 - Not useful at all, 5 - Very useful
(Q2) # Contexts generated with unnecessary details	0/10	No unnecessary details in generated contexts
(Q3) # New Rootcauses from generated contexts	2	False negatives identified by AutoARTS
(Q4) # Examples with a crucial Rootcause tag missing in classification	7/10	Crucial contributing factor missing from predictions

Table 7: **Quantitative user feedback from an expert over the effectiveness of AutoARTS across context generation and root cause classification tasks over a randomly chosen set of 10 incidents.**

6.4 Fine-tuning Language Models

R4: Can existing language models like BERT be finetuned to model software incident reports?

We conducted Masked Language Modeling(MLM) based pre-finetuning of BERT to fit our domain-specific language model, using 110K PIRs (un-tagged due to scale) from several Microsoft services. We then finetuned the models by leveraging the OCE-assigned "root cause Category" tag (which are chosen from the predefined taxonomies in Microsoft) of each of the PIRs. As mentioned in §2, OCE-assigned root cause category tags are not accurate; however, they are available at a large scale and this classification task is semantically the closest to our target classification task using the ARTS taxonomy. Table 6 shows the results for both pre-finetuning and finetuning tasks, highlighting a high perplexity of 7.57 for BERT-uncased (perplexed between choosing 8 candidate words for a blank in a given sentence) and poor classification accuracy ($\approx 35\%$) on the finetuning task. Errors in tagging of PIRs (by OCEs) coupled with lack of sufficient training data makes finetuning language models infeasible. Due to space constraints, we omit the results from other variations of BERT, but we had similar experience with them.

7 User Study

To evaluate the utility of AutoARTS, we randomly sampled 10 example incidents and the tool's generated contexts, the corresponding root cause categories and presented them to one of the leads that developed the ARTS taxonomy (by studying PIRs). These were examples that were tagged by them in the past that are not used for training any of our models. The goal of this study is to understand, for each example: (Q1) How useful the generated context is in identifying all the contributing factors that they identified, (Q2) If the generated context has extra details that are not useful for identifying contributing factors (to evaluate the succinctness of our generated contexts), (Q3) Whether the generated context can help them identify any new contributing factor that they have not identified previously (to evaluate the generalization of the model's outputs beyond accidental False Negatives in ground-truth) and (Q4) Whether the tool missed the most important contributing factors (to evaluate the importance of False Negatives).

Although we quantitatively evaluated the syntactic similarity of generated contexts to the ground truth, the developer study helps us understand if they are semantically similar and ultimately usable by a human (OCE). Similarly for Root cause classification task, the relative severity of each individual contributing factor is not identifiable from ground truth (no ranking). Q4 helps us understand if the predicted contributing factors miss any crucial tag from the ARTS taxonomy.

We quantify response to Q1 on a Likert scale of 1 to 5, where 1 meant 'not useful at all' and 5 meant 'Very useful'. Q2-Q4 were answered as a binary Yes/No, by providing clarifying responses wherever necessary for sanity check. Table 7 shows the utility of the tool based on the subject's responses to Q1-Q4. We find from the study that the contexts generated by the tool are extremely useful in identifying all the contributing factors and they are succinct enough without presenting additional information that is not useful in identifying contributing factors. In addition to this, we also found that our tool helped the subject find 2 new root cause tags that should have been assigned to these incidents in the past, highlighting the difficulty in manually sifting through postmortem reports to identify contributing factors.

At the end of the experiment, the subject was asked to answer on a scale of 1-5 (5 being very useful, 1 being not useful at all), indicating the overall usefulness of our tool to assist them in their task based on the 10 examples. The subject rated our tool 'above 4.5'. In addition to this, the subject's verbatim feedback on our tool — **'This tool is very useful from context generator perspective for the root cause classification task. From the Tags perspective, if we had 4th level for just code change related tags this is very useful for change management standards team. Need to fix the dependency tags related logic as it's defaulting to "Data Bricks". Over all I am very happy with this tool'** — highlights the utility of our context generation and the lapses in automated root cause classification. The imbalance in tag distribution over our training set resulted in misclassifying incidents with tags that do not have sufficient supporting training samples. Overall, the feedback indicates the promise for deploying the tool for practical use in assisting engineers by providing enough context from PIRs to assign root cause tags from ARTS taxonomy.

8 Related Work

Rootcause analysis of past incidents. Rootcause analysis of incidents and outages and defining taxonomies to capture their root causes has been a popular topic of study in the software engineering and systems community. We find that prior work can be categorized into two major buckets. The first category of prior work focuses on specific type of production issues such as software bugs [4, 5, 11, 19, 21, 40] or network issues [14]. The other category focuses on specific services or systems such as big-data systems [38], business data processing platform [9], high performance computing [17, 29] and a specific cloud service [13]. In contrast, we consider a large-scale cloud system consisting of many hundreds of services and *all* types of failures including hardware and software, infrastructure and application, software code and configuration, and so on. The scale of our study also differentiates us from existing studies (e.g., 152 incidents from 1 service considered in [13], 100 incidents from networking service in [14]). To the best of our knowledge, this is the most comprehensive effort of analyzing production incidents and building a fine-grained taxonomy. Prior work that propose solutions to simplify the task of actively identifying the root cause of a failure [10, 32] are orthogonal to our focus.

Text Summarization & Root cause classification. Text summarization [33] is the task of rewriting a long document into a condensed form while retaining its essential meaning, hence reducing the burden to read through lengthy documents. The most prevalent paradigms for summarization are extractive and abstractive based approaches. When generating summaries, abstractive summarization approaches [35] are typically considered as a sequence-to-sequence learning problem [24, 26, 30], whereas extractive summarization methods [12, 41] extract key sentences as summary directly from the text. In our context extraction task, we utilize the gap sentence based summarization technique not to condense the PIR context, but to extract essential text snippets describing different contributing factors. Saha et al. [28] construct a causal knowledge graph from postmortem reports but do not generate consistent root-cause tags for incidents. This can lead to ambiguous and non-uniform tagging of similar incidents, as we observed from manual tagging using different taxonomies (in Finding 2).

Prior work focused on diagnosing different kinds of incidents such as, Rex [22] suggests changes in potential misconfigurations using syntax trees, DeepAnalyze [32] identifies culprit frame in Windows Error Reporting (WER) crash stack traces, Orca [2] identifies buggy commits using differential code analysis and provenance tracking, Revelio [10] generates debugging queries for root cause analysis using logs and user reports, SoftNER [31] analyzes postmortem reports to extract entities. To the best of our knowledge we are the first to classify incident postmortems into an extensive high-granularity taxonomy.

9 Discussion

Generality of AutoARTS. Postmortems are routinely conducted in large scale production cloud systems to document learnings from incidents, similar to PIRs in Microsoft Azure (e.g., Google [14], AWS [1], and Cloudfare [7]). These postmortem reports may have different structures and content across different clous systems, but they all contain natural language descriptions of root cause diagnosis of incidents. Since the ARTS taxonomy is based on a diverse and large set of services and incidents, and AutoARTS is trained on postmortems, we believe that they can benefit other cloud systems.

Evolution of Taxonomy. Our open-sourced ARTS taxonomy captures a wide range of contributing factors, but new factors may emerge and the taxonomy may evolve. Therefore, we deliberately separate context generation and classification in AutoARTS so that new categories can be detected from the generated contexts (Table 7). When a new category is identified, the HiAGM model of AutoARTS can be finetuned for the new tags only (which takes a few minutes). We also hope others can contribute to the growth of ARTS taxonomy.

10 Conclusion

Incident postmortems are treasure troves of rich insights and retrospective analyses of them reveals actionable insights to improve reliability and availability of large-scale production systems. Unfortunately, it's not done at scale today because of the manual effort required in analyzing them. We developed a novel hierarchical and comprehensive taxonomy based on a painstaking extensive multi-year analysis of 2000+ severe incidents across 450+ Microsoft Azure services. To make this taxonomy accessible and assist engineers in analyzing postmortem reports, we proposed techniques to extract key context from postmortems and automatic classification to identify all the contributing factors for an incident and presented our findings. To the best of our knowledge, this is the largest study of production incident postmortem reports yet.

We envision that this paper enhances the audiences' understanding of contributing factors for production incidents and fosters future research by leveraging the open-sourced taxonomy for root cause labelling. Our experimental findings show promise in assisting engineers with classifying postmortem reports and we intend to fully automate this task incorporating engineers' feedback and leverage larger training datasets in future work.

Acknowledgements

We thank the ATC reviewers and our shepherd Yu Jiang for their insightful comments. Pradeep Dogga was partially supported by NSF grant CNS-1901510 and research grants from Google, Amazon and Cisco.

References

[1] aws.amazon.com. Post-Event Summaries. https://aws.amazon.com/premiumsupport/technology/pes/.

[2] R. Bhagwan, R. Kumar, C. Maddila, and A. A. Philip. Orca: Differential bug localization in large-scale services. In *13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, October 2018. Won the Jay Lepreau Best Paper Award.

[3] S. Bird. Nltk: The natural language toolkit. In *Proceedings of the COLING/ACL on Interactive Presentation Sessions*, COLING-ACL '06, page 69–72, USA, 2006. Association for Computational Linguistics.

[4] H. Chen, W. Dou, Y. Jiang, and F. Qin. Understanding exception-related bugs in large-scale cloud systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 339–351. IEEE, 2019.

[5] X. Chen, C.-D. Lu, and K. Pattabiraman. Failure analysis of jobs in compute clouds: A google cluster case study. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 167–177. IEEE, 2014.

[6] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[7] Cloudfare. Cloudfare Status - Incident History. https://www.cloudflarestatus.com/history.

[8] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[9] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, G. Goel, S. Sarkar, and R. Ganesan. Characterization of operational failures from a business data processing saas platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 195–204, 2014.

[10] P. Dogga, K. Narasimhan, A. Sivaraman, S. K. Saini, G. Varghese, and R. Netravali. Revelio: Ml-generated debugging queries for finding root causes in distributed systems. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022.

[11] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 539–550, 2018.

[12] S. Gehrmann, Y. Deng, and A. M. Rush. Bottom-up abstractive summarization. *arXiv preprint arXiv:1808.10792*, 2018.

[13] S. GHOSH, M. Shetty, C. Bansal, and S. Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *SoCC 2022*. ACM, November 2022.

[14] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.

[15] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*, pages 1–14, 2014.

[16] 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 Seventh ACM Symposium on Cloud Computing*, pages 1–16, 2016.

[17] D. Jauk, D. Yang, and M. Schulz. Predicting faults in high performance computing systems: An in-depth survey of the state-of-the-practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[18] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[19] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–530, 2016.

[20] C.-Y. Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[21] H. Liu, S. Lu, M. Musuvathi, and S. Nath. What bugs cause production cloud incidents? In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2019.

[22] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 435–448, Santa Clara, CA, Feb. 2020. USENIX Association.

[23] D. Miller. Leveraging BERT for extractive text summarization on lectures. *CoRR*, abs/1906.04165, 2019.

[24] R. Nallapati, B. Zhou, C. Gulcehre, B. Xiang, et al. Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*, 2016.

[25] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[26] R. Paulus, C. Xiong, and R. Socher. A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304*, 2017.

[27] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. 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.

[28] A. Saha and S. C. Hoi. Mining root cause knowledge from cloud service incident investigations for aiops. *arXiv preprint arXiv:2204.11598*, 2022.

[29] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7:337 – 351, 01 2011.

[30] A. See, P. J. Liu, and C. D. Manning. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*, 2017.

[31] M. Shetty, C. Bansal, S. Kumar, N. Rao, N. Nagappan, and T. Zimmermann. Neural knowledge extraction from cloud service incidents. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 218–227. IEEE, 2021.

[32] M. Shetty, C. Bansal, S. Nath, S. Bowles, H. Wang, O. Arman, and S. Ahari. Deepanalyze: Learning to localize crashes at scale. In *ICSE 2022*, May 2022.

[33] T. Shi, Y. Keneshloo, N. Ramakrishnan, and C. K. Reddy. Neural abstractive text summarization with sequence-to-sequence models. *ACM Transactions on Data Science*, 2(1):1–37, 2021.

[34] A. Vidyasagar. The art of root cause analysis. *Quality Progress*, 49(1):48, 2016.

[35] S. Xu, X. Zhang, Y. Wu, and F. Wei. Sequence level contrastive learning for text summarization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 11556–11565, 2022.

[36] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019.

[37] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 1480–1489, 2016.

[38] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. 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, 2014.

[39] J. Zhang, Y. Zhao, M. Saleh, and P. Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In *International Conference on Machine Learning*, pages 11328–11339. PMLR, 2020.

[40] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan. Understanding and detecting software upgrade failures in distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 116–131, 2021.

[41] M. Zhong, P. Liu, Y. Chen, D. Wang, X. Qiu, and X. Huang. Extractive summarization as text matching. *arXiv preprint arXiv:2004.08795*, 2020.

[42] J. Zhou, C. Ma, D. Long, G. Xu, N. Ding, H. Zhang, P. Xie, and G. Liu. Hierarchy-aware global model for hierarchical text classification. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1106–1117, 2020.

Avoiding the Ordering Trap in Systems Performance Measurement

Dmitry Duplyakin *Nikhil Ramesh* *Carina Imburgia*[*]
Hamza Fathallah Al Sheikh *Semil Jain* *Prikshit Tekta*
Aleksander Maricq *Gary Wong* *Robert Ricci*

University of Utah [*]University of Washington

Abstract

It is common for performance studies of computer systems to make the assumption—either explicitly or implicitly—that results from each trial are independent. One place this assumption manifests is in experiment design, specifically in the order in which trials are run: if trials do not affect each other, the order in which they are run is unimportant. If, however, the execution of one trial *does* affect system state in ways that alter the results of future trials, this assumption does not hold, and *ordering must be taken into account* in experiment design. In the simplest example, if all trials with system setting *A* are run before all trials with setting *B*, this can systematically bias experiment results leading to the *incorrect* conclusion that "*A* is better than *B*" or vice versa.

In this paper, we: (a) explore, via a literature and artifact survey, whether experiment ordering is taken in to consideration at top computer systems conferences; (b) devise a methodology for studying the effects of ordering on performance experiments, including statistical tests for order dependence; and (c) conduct the largest-scale empirical study to date on experiment ordering, using a dataset we collected over 9 months comprising nearly 2.3M measurements from over 1,700 servers. Our analysis shows that ordering effects are a hidden but dangerous trap that published performance experiments are not typically designed to avoid. We describe OrderSage, a tool that we have built to help detect and mitigate these effects, and use it on a number of case studies, including finding previously unknown ordering effects in an artifact from a published paper.

1 Introduction

Systems performance analysis typically involves running a series of trials and then calculating statistical measures (such as mean or median) from the performance data collected. These measures are used to conclude that one system is, on average *X*% faster than another, that the addition of a new feature does not have a statistically-significant impact on performance [12, 16], or that software scales well to large problem sizes. One of the most fundamental assumptions of this kind of analysis [36] is that trials are *independent*; in particular, that each trial is unaffected by prior trials in the series. If this assumption does not hold, it can systematically bias results and alter or even invalidate conclusions drawn from them.

Typical systems research work does not take ordering into consideration as part of experiment design. This can lead to violations of the independence assumption.

The problem is especially pernicious because there is not one, or even a few, root causes behind performance-affecting state that carries over between trials. In the highly complex environment of a modern computer system, there are a large number of hardware and software components whose state can be carried over from one trial to another [26]. These include caches [8], data layout in RAM and on disk [22], application and operating system tuning parameters [20, 41], and even temperature (with consequences such as thermal throttling [3, 13]). The systems under test themselves can, intentionally or unintentionally, make changes that persist between trials, such as changes to software packages, global system configuration, environment variables [26], or files.

Thus, while the question of *why* order matters is important, it is highly specific to the software being tested, the hardware it is run on, and the design of the experiment. Before "*why*" can be considered, there is the more fundamental question of *whether* the order matters for a specific experiment. In many cases, knowing that order-dependent performance *exists* can itself be an interesting result because it indicates some unexpected property of the software or system under test. Therefore, eliminating it entirely through experiment design is not always even desirable.

In this paper, we formulate a systematic approach to analyzing whether the order of trials within an experiment affects results. We use this method to collect and analyze a large new performance dataset that we collected on over 1,700 servers over a period of 9 months and show that experiment order is a factor that cannot be neglected. We find that for the selected benchmarks the order can bias performance by 50% or more and potentially alters conclusions in 72% of cases.

Order is acknowledged to have some level of impact in the literature [1, 26]. However, we show this acknowledgment has not translated into experiment design *in practice*. We conducted a survey of three major systems conferences and found that it is exceedingly uncommon to discuss experiment ordering in these papers. Furthermore, we examined the artifacts for the papers and find that they are not designed to detect or avoid ordering effects. To help relieve this situation, we contribute OrderSage, a tool that helps experimenters with both

the orchestration and analysis aspects of experiment ordering.

In this paper, we make the following contributions:

- We perform a literature survey of top-tier systems conferences (Section 2), showing that experiment order is reported as part of experiment design in fewer than 10% of papers. We also analyze these paper's artifacts, and show that this neglect extends to the way experiments are run in practice: more than 94% of artifacts run experiments in either a single fixed order or do not specify an order.
- We develop a methodology (Section 3), using established statistical tests, for determining whether results are order-dependent and narrowing down specific experimental tests that are particularly affected.
- We collect and publicly release a large, first-of-its-kind dataset for studying the impact of ordering on performance experiment results (Section 4). This dataset contains the results of over 2.3M trials run in a variety of different orders.
- We analyze this dataset using our methodology (Section 5) and show that ordering can make a significant difference, even to the level of potentially changing conclusions. This provides strong evidence for the claim that systems researchers should consider order in their experiment design.
- We developed and release OrderSage, a tool that easily applies our methodology to performance experiments (Section 6). OrderSage embodies both a mechanism for randomizing experiment order and analyzing its effects. To demonstrate its use, we present case studies (Section 7) applying it to the performance test-suite for `memcached` [2, 7, 21], and to NPBench [43]. We also use OrderSage on one of the artifacts from our literature survey and find a previously-unknown ordering effect in it.

We cover related work in Section 8 and conclude in Section 9.

2 Literature and Artifact Survey

To evaluate the extent to which ordering effects are taken into account in practice in the systems research literature, we conducted a survey involving the OSDI '21, SOSP '21, and EuroSys '22 conferences. We selected these three conferences because they ran *Artifact Evaluation Committees* (AECs), meaning that we were able to look at both what *papers say* about ordering and what the artifacts (code, scripts, etc.) *actually do*.

We had two inclusion criteria for our survey. First, the papers need to have received all three AEC badges (Available, Functional, and Reproduced)—this lets us know that not only did the paper have an artifact submitted, but that the artifact is complete. Second, the papers need to base their main claims

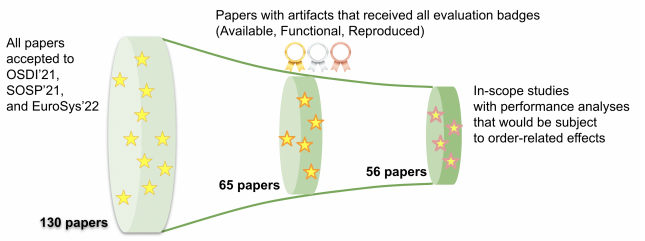


Figure 1: Paper selection for our literature and artifact survey.

on a set of performance metrics (e.g., runtime, latency, bandwidth, etc.) executed on real (not simulated) systems. Under these criteria, we ended up with 56 papers out of the three conferences' 130 papers, as seen in Figure 1.

Following the selection and filtering phase, we performed the survey in two passes; each paper/artifact was reviewed by a different reviewer in each pass, with the goal of countering individual reviewers' biases. Table 1 presents the results from both passes as well as the agreement between the two passes. It is worth noting that the spread is larger for the artifact analyses because they required more investigation than reading the evaluation sections of the surveyed papers. Regardless, we consider all observed relative agreement numbers to be high enough to serve as a convincing basis for our conclusions. As detailed below, our artifact review provided much more insight into how the studies were run compared to the information in the papers alone.

Do papers specify the ordering of their experiment design? Only 4 out of 56 papers (7%) clearly stated the order in which the corresponding performance experiments were run. This percentage is not surprising, because space constraints lead authors to focus on describing the factors that are key to their work instead of latent factors such as the order of execution. Of special note is that EuroSys '22 allowed artifact description appendices, which we considered as part of the paper rather than part of the artifact. This is where we found most of the ordering-related information; these appendices allowed authors to detail steps in their evaluation workflows, leaving no ambiguity about the orderings.

Do papers describe their inter-experiment reset procedures? Between 4 and 10 papers, or 7–18%, described reset procedures for ensuring that subsequent trials are not potentially impacted by the preceding tests. Such procedures included clearing caches, running warmup tests, rebooting hardware, and launching new cloud instances, among others. Similar to our conclusion about the order information, the reset specification was scarce in the studied papers.

What order do artifacts execute experiments in? Because papers do not tell us much about what order is used for experiments, we examined the artifacts themselves. 36 to 37 artifacts (64–66%) use a fixed-order experiment design. This was typically implemented by providing a "run all" script

Table 1: Results of studying 56 papers and the corresponding artifacts.

Attribute being tested	1st pass	2nd pass	Match b/w passes
Paper explicitly describes an order of experiment execution	4 (7%)	4 (7%)	93%
Paper describes a reset procedure to be run between experiments	4 (7%)	10 (18%)	82%
Artifact's primary experiment execution order:			63%
fixed	36 (64%)	37 (66%)	
undefined	17 (30%)	17 (30%)	
parallel	3 (5%)	2 (4%)	
Artifact runs a reset procedure between experiments	27 (48%)	16 (29%)	73%

that iterates through the studied algorithms or configuration options in sequence with no randomization. In other cases, a specific order was documented in the repository's README files. Many other artifacts (17, or 30%) provided instructions on how to run individual groups of tests (e.g., for specific figures and sections in the papers) but did not specify any sequence between them—we categorized their orderings as undefined. Another small class of studies (2–3 artifacts) used parallel execution, where tests were run concurrently on multiple worker machines or cloud instances and therefore can be considered to run each test in its own clean environment. We have not identified any artifacts that implemented a randomized ordering, or which clearly showed explicit attention to ordering concerns. To summarize, 53–54 artifacts out of 56 (94–96%) used undefined or fixed orderings, both of which can be questioned from the presentation and experiment design perspectives. Expanding on the latter case, we show in this study that fixed-order experiment designs have potential to introduce adverse bias in performance analysis.

Do artifacts use inter-experiment reset procedures? Between 16 and 27 artifacts (29–48%) ran identifiable procedures to reset the system to a known state between experiments. Finding these procedures in the code is a non-trivial and time-consuming process, which explains the spread between the results in the two survey passes. While we did find reset procedures in up to half of the artifacts, it is concerning that the other half of the artifacts did not manifest any reset procedures. While it may not matter for some of the studies because of the nature of their performance analysis, there is a chance that for a subset of them it may be an oversight causing undesirable effects on their conclusions.

From the survey, we learn that the literature does not make order an explicit part of experiment design, and we do not see evidence that ordering issues are explicitly addressed. In the remainder of this paper, we show how this can constitute a trap for experimenters and discuss how to avoid this trap.

3 Analyzing Order Dependence

In this section, we detail the procedure we have designed to find order dependence in performance experiments. There are two primary outputs from this procedure: first, it reports whether the statistical distribution of performance results differs when run using *fixed-order* and *random-order* experiment designs—that is, whether the order has an effect on the experiment results. Second, it reports whether these differences are potentially large enough to *change inferences*—that is, whether it is possible for ordering effects to be large enough that the conclusions drawn from an experiment could change based on the execution order.

For consistency, we use the following terminology in this section and throughout the remainder of the paper:

Test: A test is an individual unit of the system under evaluation. A test typically represents an individual benchmark or an application with a specific configuration or input.

Trial: A trial is an execution of a test. The outcome of a trial is a single-metric performance assessment, such as runtime, throughput, latency, etc. Multiple trials of the same test typically exhibit variations in performance stemming from the nondeterminism intrinsic to the test itself or the system used for benchmarking.

Run: A run is a set of trials, in a particular order, of *all* tests in series. Conceptually, one could (and often does) report the results from a single run in a single order.

Experiment: An experiment is a collection of one or more runs done for the purpose of reaching a conclusion about the system(s) under evaluation; typically, such a conclusion will be reached by comparing results of the trials associated with different tests. The order of trials within the runs of an experiment is part of the *experiment design* and is referred to as the *experiment order*.

An outline of the method is shown in Algorithm 1; below, we go through each step in detail.

❶ Select a "Baseline" Order Select an ordering of trials that will be used for "fixed order" runs. The order itself is not important; the order in which trials have been run in the past, or a "natural" order (such as by increasing parameter value) is sufficient. This does not need to be a "correct" order: it will act as the control against which we test random orderings.

❷ Define a "Reset to Clean State" Procedure Each run (series of trials) should start from a clean state, such that

Algorithm 1 Order-Dependence Test

Input T: List of trials in baseline order ▷ ❶
Input R: Reset procedure ▷ ❷
Input N: Number of repetitions
Input α: Desired family-wise error rate (commonly 0.05)
1: **for** $n = 1, \ldots, N$ **do** ▷ ❸
2: Execute R
3: **for all** $t \in T$ **do** ▷ Run trials in baseline order
4: fixedOrderResults$[t][n] \leftarrow$ Execute t
5: **end for**
6: Execute R
7: **for all** $t \in$ RandomlyPermute(T) **do** ▷ Run trials in random order
8: randomOrderResults$[t][n] \leftarrow$ Execute t
9: **end for**
10: **end for**
▷ ❹
11: **for all** $t \in T$ **do** ▷ Calculate p-values for distribution comparison
12: $p_{KW}[t] \leftarrow$ KruskalWallis(fixedOrderResults$[t]$, randomOrderResults$[t]$)
13: **end for**
14: $\alpha_{BC} \leftarrow \alpha / length|T|$ ▷ Use Bonferroni corr. for multiple comparisons
15: **if** $\exists t \in T \mid p_{KW}[t] < \alpha_{BC}$ **then**
16: **return** true ▷ Order matters for 1 test $\rightarrow$ it matters for the experiment
17: **else**
18: **return** false
19: **end if**

Algorithm 2 CI Overlap Test

Input fixedOrderResults, randomOrderResults from Algorithm 1
Input t: test to check
1: (fLow, fMedian, fHigh) $\leftarrow$ RankBasedCI(fixedOrderResults$[t]$) ▷ ❺
2: (rLow, rMedian, rHigh) $\leftarrow$ RankBasedCI(randomOrderResults$[t]$)
3: **if** (fLow $>$ rHigh) $\vee$ (fHigh $<$ rLow) **then**
4: **return** Case 1 ▷ Inference *does* change
5: **else if** (fLow $<$ rMedian $<$ fHigh) $\wedge$ (rLow $<$ fMedian $<$ rHigh) **then**
6: **return** Case 2 ▷ Inference likely does *not* change
7: **else**
8: **return** Case 3 ▷ Inference may or may not change
9: **end if**

state left over from earlier runs will not affect performance results of the next run. In many cases, this will be much more expensive than running the benchmarks themselves: for the purposes of the experiments in this paper, this procedure is a reboot of the server on which the benchmarks are executed. This might instead consist of restarting a server process, provisioning fresh VMs, clearing storage devices, etc.

❸ Run in Both Fixed and Random Orders We execute a series of runs. Each run consists of the same set of trials, and each trial may comprise of multiple invocations of the system under evaluation in order to increase statistical significance. For half of our runs, trials are run in the fixed baseline order; in the other half, the order is randomly permuted (separately for each run). Between runs, the environment is reset to a clean state using ❷. Since the evaluation might take long enough that time-varying effects (such as hardware degradation) could be observed, fixed (Lines 3–5) and random order (Lines 7–9) runs are interleaved to avoid bias. The outcome of each run is a set of performance results, one from each trial, with the units being the "natural" units for the tests, e.g., seconds for runtime, MB/s for bandwidth, etc. The experimenter should complete a sufficiently sized set of runs (Line 1) to provide the desired statistical significance in the subsequent steps.

❹ Compare Distributions The next step is to compare the samples obtained from the fixed- and random-order runs. The intuition behind this step is that if the two sets of samples come from the same statistical distribution, it can be said that the order does not change the distribution, and thus does not matter. If they come from different distributions, then the order does indeed matter.

To avoid assumptions of normality (which have been shown to rarely hold for computer systems performance results [22]), we use the non-parametric Kruskal-Wallis test (Lines 11–13). This distribution-comparison test produces a p-value indicating the likelihood of observation assuming the null hypothesis (i.e., that both samples come from the same distribution). This should be performed *for each test*, longitudinally across all runs: the two populations are (a) the outcomes for all trials (executions) of the test from fixed-order runs, and (b) the outcomes of all trials for the same test from random-order runs. Thus, we are looking at whether a particular test's performance differs based on where its trials occur in the runs that are differently ordered.

For each test, compare the p-value produced by Kruskal-Wallis with a threshold chosen to provide the confidence level desired; we aim for a family-wise error rate of $\alpha = 0.05$ (95% confidence) as is common in such tests. Because we perform a potentially large number of comparisons, the problem of multiple comparisons [25] arises; we apply the Bonferroni correction [9] (Line 14) to obtain the per-test thresholds (α_{BC}) required for multiple comparisons to reach the target family-wise confidence level. This correction scales the thresholds down (making them stricter) in proportion to the number of comparisons made.

If the p-value is above the threshold, we cannot reject the null hypothesis, and therefore conclude that both samples could have come from the same distribution—the order *likely does not matter*. If the p-value is below the threshold, we reject the null hypothesis and conclude that a single distribution would be highly unlikely to yield the observed samples—*the order of the tests **does** matter*.

We note that it is possible, and in our experience common, that different tests within an experiment produce different results at this step. This could indicate that some tests are affected by what runs before them and others are more robust in this respect. Overall, however, as long as *any* test shows order-dependence, this indicates that the experiment design as a whole needs to be aware of ordering (Lines 15–19).

❺ Compare Confidence Intervals A typical experiment setup in performance analysis is to ask whether there is a difference in performance between two systems. A situation

particularly important to avoid is one in which inferences from the experiment could *change* depending on the ordering, in turn leading to a change in the *conclusions drawn*. We look for such situations by comparing confidence intervals [12] (CIs) as shown in Algorithm 2. CIs can be compared *within* tests (e.g., comparing fixed and random orders, to determine whether order changes the median computed), or *across* tests (e.g., comparing two or more tests and checking whether different performance is observed.)

The outcome of this test tells us something related to, but distinct from, the Kruskal-Wallis test. Kruskal-Wallis tells us whether the distributions differ, but not directly whether they differ enough to change conclusions in a significant way. Looking at the effect size (detailed in Section 5.1.1) gives us a sense of the latter, but the CI test answers it directly. Recall that a CI is an estimated interval we expect to include the true value of a population measure [12]. For instance, for the 95% CI of the median (the interval we use), we expect that in a collection of many such intervals, 95% of our estimates would contain the true population median.

We use rank-based CIs estimating the population median [17] to avoid assumptions of normality. This comparison results in three possible cases (visualized in Figure 5):

Case 1: The CIs for the fixed- and random-order runs do not overlap. In this case, we can have high confidence that we would expect to compute different medians depending on the order. This is a **red flag**, and indicates that we could come to different conclusions based on the order.

Case 2: The median for at least one of the two samples lies within the CI for the other population. If this is the case, given one population, we could have potentially arrived at the *other* observed median, and we conclude that our conclusions likely would not change based on order.

Case 3: In the final case, the CIs overlap, but both medians are outside the other group's interval. This case is inconclusive, and requires more careful analysis to determine if it could change conclusions. Still, it is a potential sign that more careful experiment design is needed.

4 Dataset and Data Collection

To study performance effects at a large scale, we have collected a dataset covering nearly 2.3 million executions (*trials*) of 25 benchmarks on 1,700 machines over a period of nine months. Many benchmarks were run in multiple configurations, such as on different sockets or with different CPU frequency settings, resulting in multiple *tests* per benchmark application. This data was collected across more than 9,000 *runs*. We released this dataset as part of this paper's artifact: `https://github.com/ordersage/paper-artifact`. Collection of the dataset covers the first three steps of the method described in Section 3; we cover the rest of the steps in the this section.

This dataset focuses on *low-level measurements* of CPU and memory performance through the use of standard benchmarks, in particular STREAM [23], the NASA Parallel Benchmarks [27] (NPB), and Reece's memory benchmarks [30, 31]. We have additional benchmarks of disk and network performance, but leave analysis of them to future work. Our case studies in Section 7 have examples of our methodology applied to higher-level applications.

4.1 Environment

We collected our data by running experiments in CloudLab [5], a public testbed for research use. CloudLab has a variety of different types of server hardware [37], and we ran our experiment across 13 different server types. We considered each configuration of each benchmark on each node type as constituting its own test for the purposes of this analysis: thus, we have 1,880 different collections of corresponding trials to compare. CloudLab is an attractive platform for this work, as it has previously undergone study to quantify and calibrate the level of variability across different hardware in the platform [22].

Servers in CloudLab are allocated at a *bare-metal* level to one user at a time. Disks used are all local to the server, and for this paper, we do not consider the network or other shared resources. Thus, our benchmarks were not affected by any other simultaneous users of the servers in question or the CloudLab system as a whole, and did not have any artifacts due to virtualization. We believe our dataset to be robust with respect to time-varying, location-dependent (e.g., environmental/temperature), and micro-architectural factors: we gathered this data over a period of months; CloudLab nodes are in three different geographically distant data centers; and they encompass a variety of processor and memory technologies.

4.2 Baseline Order ❶

The baseline order that we use is a "natural" one that groups benchmarks from the same suite (e.g., NPB [27]) together, and reflects the order in which we added the suites to our experiment setup. This reflects the type of order that a systems experimenter would be likely to arrive at in the process of developing scripts to run their experiments.

4.3 Reset Procedure ❷

The reset procedure we use is a fresh load of the operating system and clean boot of the host on which the experiments are run. This means that each run sees, as much as possible, the "pristine" state of a just-booted machine, not affected by any software or configuration changes made by prior users.

It is important to note that we do not claim this clean state to be *correct*: we do not claim that the results of a trial gathered under these conditions are more "valid" than results after the machine has been running for some time. It is possible for boot-time effects to alter results, and for some tests, a scenario in which a machine has been booted and active for a long period of time may be more *realistic*. What we *do* claim about this procedure is that we can be confident that all runs *started* from the same state. Therefore, it can tell us if the order of trials within the run affected results.

4.4 Running in Fixed and Random Orders ❸

Our data collection framework allocates machines in CloudLab on which to perform runs. For each run, it randomly chooses—with equal probability—to execute trials in our *fixed baseline* order or a *random order*. This procedure ensures that we *interleave* baseline and random runs, running them in approximately equal proportion throughout the entire time period to avoid a systematic bias in one direction or the other due to potential changes in the facility over time. If the random order is chosen, the framework shuffles the list of all trials for that run. The framework records this order information for use in future analysis.

5 Analysis Applied to Our Dataset

We now describe how we analyze the order-dependence of the performance results gathered in Section 4. This section covers steps ❹ and ❺ from the method described in Section 3.

The nature of our data collection adds another dimension to our *tests*, and thus we adopt terminology used elsewhere in the literature [22] for clarity. Because CloudLab contains servers of many different types, each of the *tests* we define will be executed on 13 different hardware types—each different hardware type may have a different processor, different amount of RAM, etc. We refer to a combination of *{test, hardware type}* as a *configuration*, where the *test* itself is a combination of *{benchmark, settings}*. For example, the STREAM benchmark run in its `COPY` mode on a server of type `m400` represents one configuration; STREAM in `COPY` mode on a server of type `c6520` is another; and STREAM in `SCALE` mode on an `m400` would be a third. In total, we have 1,880 such configurations. Results from *trials* executed under a particular *configuration* across all *runs* are grouped together: our primary comparison of interest is whether the same configuration produces different results when run as part of differently-ordered runs. It is worth noting that we do not expect results from different configurations to be independent, and do not analyze them as such: there is strong likelihood, for example, that STREAM in `COPY` mode exhibits similar order-dependent performance effects to STREAM in `SCALE` mode. The value derived from running so many configurations is that it helps to make our results robust with respect to many different programs, settings for those programs, and types of hardware.

We analyze data from our memory and CPU benchmarks as separate experiments: this avoids mixing results from performance tests with very different goals, and offers interesting insight into how the effects of ordering can differ depending on the main resource being exercised.

5.1 Comparing Distributions ❹

The next step in our method is to compare the distributions of performance results for each configuration when run in fixed vs. random orders.

5.1.1 Memory Benchmarks

The left side of Figure 2 plots the p-values for all 1,198 configurations of memory benchmarks. For this test, the Bonferroni-corrected α_{BC} ($n = 1,198$) is 4.2×10^{-5}. Configurations are sorted on the x-axis according to the effect size (discussed below). As can be seen from the figure, most (1,042, or 87%) of the configurations fall well below the α_{BC} threshold, showing clear evidence of performance effects due to ordering.

To strengthen our analysis, we calculated the *effect size* for each pair of compared samples. This measure is not meant to replace the p-values but rather should complement them [40]. While the statistical tests indicate that the probability of the sampling error causing the observed performance difference may be low, measuring the effect size helps us understand the scale of the difference between the groups.

We calculate the effect size for each statistical test. The larger the effect size, the larger the estimated difference between the populations being compared; a small effect size can indicate that even when there is a statistically significant difference revealed by a p-value, it may be small enough not to be of *practical* importance. To align with the Kruskal-Wallis test, we use the non-parametric formulation of the effect size η^2 that is defined using the H-statistic [4]. In statistics terms, η^2, which yields values between 0 and 1, estimates the fraction of variance in the dependent variable that can explained by the independent variable. The review article [40] provides additional context and includes the formula for η^2 calculation.

η^2 values are plotted in the right side of Figure 2. Past the first few hundred configurations, η^2 becomes larger indicating that the difference between the fixed-order and random-order results becomes larger. This is also the exact region in which $p < \alpha_{BC}$, which indicates significance. It is worth noting that we do not compare η^2 with arbitrary thresholds but rather observe its growth across the range of the tested configurations for comparison purposes; from this standpoint, it is assessed similarly to how we interpret percentage differences in Section 5.1.3.

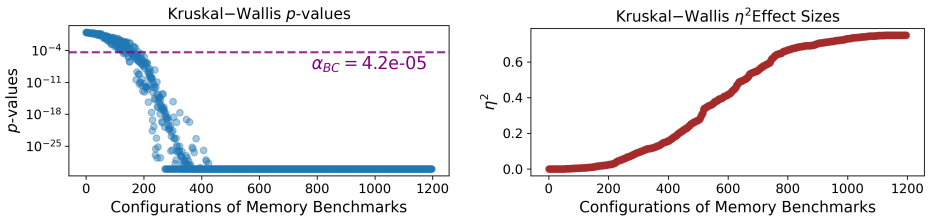


Figure 2: Kruskal-Wallis p-values and effect sizes for memory benchmarks, sorted in order of increasing Kruskal-Wallis η^2 effect size. The horizontal line at the bottom of the left plot comes from rounding small values up to 10^{-30} for display.

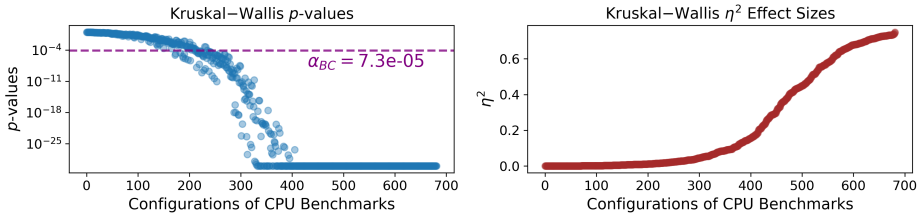


Figure 3: Kruskal-Wallis p-values and η^2 effect sizes for CPU data. The plotting is as described for Figure 2.

5.1.2 CPU Benchmarks

Figure 3 shows the Kruskal-Wallis p-values and effect sizes for our CPU benchmarks. For these comparisons, α_{BC} ($n = 682$) is calculated as 7.3×10^{-5}. As with the memory benchmarks, most configurations fall well below the α_{BC} threshold, indicating that the order in which they are run makes a difference. The most observable distinction between the CPU tests and the memory tests is the shape of the effect size curve: while there are still some configurations that have large effect sizes, there are fewer of them. There are a larger number of configurations that reach statistical significance in the p-value test but have an effect size small enough that it may not have a practical impact. This demonstrates the need to look at both significance tests *and* effect sizes.

Table 2 summarizes the observed Kruskal-Wallis p-values. From this table, we can clearly conclude that the order of experiments matters for the selected microbenchmarks. This effect appears to be more pronounced for memory benchmarks which have a higher ratio of configurations with $p < \alpha_{BC}$.

5.1.3 Relative Difference

Most computer systems studies report their results not in terms of effect sizes but in terms of absolute or relative differences between several alternatives. To align our analysis with our research community, we also looked at the order effects in terms of percentage differences (representing the difference between the mean fixed-order result and mean random-order result, divided by the mean-fixed order result):

$$\Delta_{\%} = \frac{\mu_{fixed} - \mu_{random}}{\mu_{fixed}} \times 100\%$$

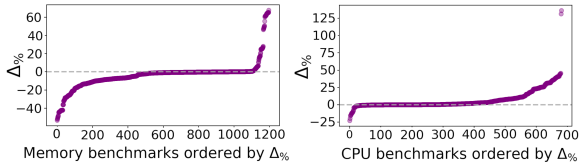


Figure 4: Percent Difference for Memory & CPU benchmarks.

Figure 4 shows the observed $\Delta_{\%}$ values for memory and CPU benchmarks. The configurations are sorted on the x-axis by $\Delta_{\%}$ values, and the y-axis is $\Delta_{\%}$ for a particular configuration. The range of $\Delta_{\%}$ values gives a sense of the magnitude of the studied order-related effects.

Our memory data is measured in throughput, so the higher the value, the better the performance—since we calculate $\Delta_{\%}$ as fixed order performance minus random order performance, a positive $\Delta_{\%}$ indicates that the *fixed* order had better performance than the randomized order. A negative $\Delta_{\%}$ means the *randomized* order performed better. Conversely, our CPU data is measured as execution times, so *lower* values mean better performance: for these, *positive* values mean that the *randomized* experiment design results in better performance.

From these figures, we can see that both memory and CPU benchmarks have some effects that would be considered large enough to affect results. Though they have similar absolute average $\Delta_{\%}$ values (8% for memory, and 7.3% for CPU), the details of their curves are quite different. Both have some configurations that are faster in random order and some that are slower, but the magnitudes and shapes of the curves differ.

Table 2: Configuration classification showing whether there is difference between fixed and random orders or not. The comparisons used Kruskal-Wallis test with Bonferroni-corrected $\alpha_{BC} = 4.17 \times 10^{-5}$ for memory and $\alpha_{BC} = 7.33 \times 10^{-5}$ for CPU comparisons.

Benchmark Type	Kruskal-Wallis $p < \alpha_{BC}$	Kruskal-Wallis $p > \alpha_{BC}$	Total
Memory	1042 (86.97%)	156 (13.02%)	1198
CPU	475 (69.65%)	207 (30.35%)	682

5.2 Comparing Confidence Intervals ❺

In Section 5.1, we showed that experiment order does impact performance using statistical tests and percentage difference. In this section, we look at whether the bias caused by experiment order can result in incorrect conclusions. Answering this question will allow us to establish whether a researcher should consider experiment order while analyzing performance.

Figure 5 shows each of the three cases from the confidence interval overlap test, using examples drawn from our memory benchmarks. The *x*-axis represents the experiment order. The *y*-axis is the rank-based 95% confidence interval of the median performance for each order, with the shaded region representing the interval and dashed lines extending the interval limits to the full width of each figure for comparison. The diamond represents the median value. Case 1 indicates that the conclusion *would* change based on order, Case 2 indicates that it is *unlikely* to do so, and Case 3 is *inconclusive*.

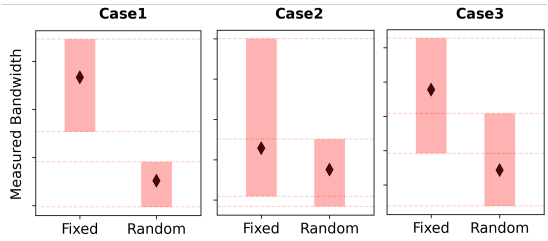


Figure 5: Examples of CI arrangements. The plots are created based on actual measurements for three memory tests; scales are different for them. Red vertical bars are rank-based non-parametric 95% CIs for medians, and ♦—median estimates.

For both memory and CPU benchmarks, we found that most configurations fell into Case 1, meaning that order could change conclusions. This can be seen in Figure 6. The effect is more pronounced for memory benchmarks, where 81% of the configurations fall into Case 1, than for CPU benchmarks, for which only 56% fall into Case 1. Overall, 72% of all configurations are in Case 1. From these results, it becomes amply clear that one can arrive at an incorrect conclusion by merely modifying the experiment order. A performance analysis needs to consider order to ensure accurate results.

6 Automating Experiment Order Testing

We have shown that order can be important in experiment designs. However, it can be difficult for experimenters to rethink their performance experiments to account for this factor. To this end, we have developed OrderSage, a tool that enables experimenters to follow the methodology from Section 3 in their experiments with minimal effort.

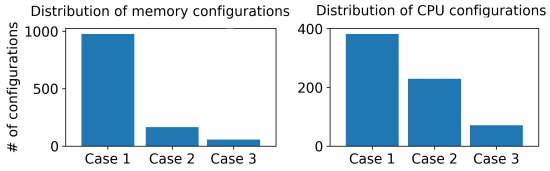


Figure 6: Three CI cases for memory and CPU tests.

6.1 Motivation

The data collection efforts described in Section 4 can be characterized as *long-term*, *extensive*, and *requiring robust infrastructure*. The first refers to the fact that we collected the studied measurements over the period of 9 months. The second indicates that we benchmarked a large pool of hardware types, used numerous tests, and studied many unique permutations of commonly tuned benchmark and system parameters. The third stresses that experimentation of this kind requires reliable computing resources and software for orchestrating test execution, gathering and storing results, etc. We met these requirements using CloudLab hardware, the testbed's programmatic interface built upon `geni-lib` Python package [38], and a set of custom scripts developed for orchestration [39].

This is in contrast to most studies, which describe *short-term* and *focused* analysis efforts, where experimenters thoroughly study subsets from many combinations of tunable parameters and gather the needed results in a limited timeframe. In such settings, the emphasis often is on demonstrating that one algorithm or hardware implementation is better than the alternatives and on characterizing its observed gains. To arrive at such conclusions, experimenters need to make sure that their measurements are not significantly impacted by the order-related effects. In the simplest cases, this means that if they were to run the same sets of tests in different orders, their conclusions would remain valid. We note that in many cases in which order-dependent performance is discovered, this may indicate unexpected behaviors in the system, and is itself an interesting finding.

Aiming to support such focused experimentation, we design OrderSage with the target user in mind who is an experimenter collecting performance data for publication (such

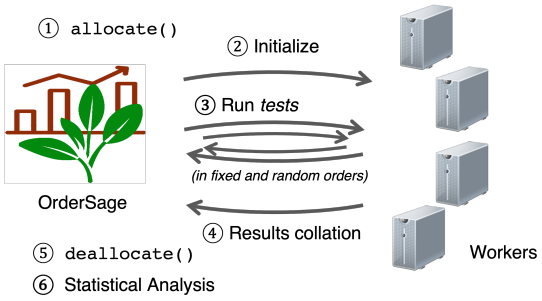


Figure 7: The main operation of OrderSage.

as in a research venue) or for decision-making (such as in a production computing environment.) We assume that such a user can script the execution of each test (e.g., in *shell* or *Python*), and thus OrderSage's primary responsibilities are to execute a number of runs with the scripted tests in different orders, collect and store test results, and analyze the results through the lens of possible order effects. Below, we describe the key components of OrderSage's experiment orchestration.

6.2 Implementation

OrderSage is implemented as a set of Python modules and is available at `https://github.com/ordersage/ordersage`. Its operation can be described using the following terms:

Controller: A machine that facilitates experimentation on a remote node or nodes and performs the statistical analyses. This is the primary place where OrderSage's code runs.

Worker: A machine that executes an experiment that consists of several fixed and random runs. Workers are accessed (through `ssh`) and controlled by a single controller node. One controller can make use of one or more workers.

Results: Each trial produces one performance result in the form of a floating-point number. Multiple performance metrics should be treated as separate results. OrderSage collects these results from the worker(s) and stores them on the controller for analysis.

Figure 7 shows a high-level system diagram that includes the key processes being orchestrated by OrderSage:

① **allocate():** Support is provided for use of worker nodes that are either pre-allocated by the user or reserved on CloudLab testbed. Outside of CloudLab, any node—local or remote—into which the user can `ssh`, run the tests (including installing software, if necessary), and execute the reset procedure can be used. Analogous `allocate()` routines can be implemented for commercial clouds using the APIs they provide.

② **Initialize:** Worker nodes are readied via an initialization script provided by the user. This typically includes installing and configuring the software under evaluation. During initialization, a baseline order is selected (Step ❶ from the method described in Section 3) by running a user-provided script that produces the commands for each test. OrderSage defaults to rebooting a worker node as the "clean state" reset procedure (Step ❷ from the method). However, this behavior can easily be redefined by the user.

③ **Run Tests:** After initialization, tests are run according to Step ❸ of the method. Each run consists of an execution of all trials in some order. Half of the runs are performed in the baseline order, and half in an order that is randomly shuffled (separately for each run); these orders are interleaved. The reset procedure is executed in between each run.

④ **Collect Results:** Performance measurements are collected as outlined in Step ❸ of our method and saved as raw data from each trial. Additionally, metadata such as random orderings, machine environment information, execution times, and `stdout` outputs are saved for each run.

⑤ **deallocate():** If the worker(s) were `allocated()`ed in ①, (such as with our CloudLab integration), OrderSage will handle the deallocation of nodes at the end of the experiment.

⑥ **Statistical Analysis:** Results are analyzed according to Steps ❹ and ❺ from the method described in Section 3. The results of all statistical analyses are saved on the controller node. If multiple worker nodes were used, the necessary result aggregation will take place. A final comparative step will provide a high-level overview of the combined vs. individual experiment analyses.

Running OrderSage is detailed in Appendix A.

7 Case Studies

We used OrderSage to facilitate the methodology proposed in Section 3 for three case studies. Our goals were to demonstrate its use on common benchmarks and application test suites and investigate the impact of test order on the results of these cases. All case studies were executed on CloudLab servers of the `xl170` hardware type [37]. For these experiments, which occurred over 24-48 hours, each experiment was run on a single worker. All data from these experiments are included in the released artifact: `https://github.com/ordersage/paper-artifact`.

7.1 `memcached` Benchmark Suite

This case study uses `memcached`'s own benchmark suite, which is designed to mimic the process of reporting perfor-

Table 3: Test results for the `memcached` experiment. We use Bonferroni correction with $n = 3$ and $\alpha_{BC} = 0.0167$ (providing a family-wise error rate of 0.05). The Kruskal-Wallis p-values are shown, as are their interpretation relative to α_{BC}: the column contains • if the null hypothesis can be dismissed or ○ if it cannot. $\Delta_{\%}$ is calculated as in Section 5.1.3 and the CI cases are as defined in Section 3.

Test	KW p-value	KW test	$\Delta_{\%}$	CI case
`cmd_set`	0.49	○	0.3	2
`cmd_get`	0.74	○	−0.2	2
`get_hits`	0.00009	•	5.3	3

mance numbers for this application. `memcached` [2] is an efficient and widely used in-memory key-value store, and its associated `mc-crusher` [24] benchmark suite includes a variety of scripts designed to exercise a server instance and measure its performance.

The `mc-crusher` documentation specifies "You should start a fresh `memcached`", and includes a series of three tests (`cmd_set`, `cmd_get`, and `get_hits`) in its included `sample` configuration file, executed serially in that order; accordingly, we start `memcached` after the reboot in our standard Step ❷ reset procedure, and perform those same three tests in each of our runs. We follow the same ordering of trials in the fixed case, with a single instance of `memcached` for all trials (following the `mc-crusher` distribution exactly). We increase the sample duration to 60 seconds per test (to reduce the influence of noise on each sample) and permute the order of the trials in our random runs to check our hypothesis that ordering affects the observed results, but otherwise do not modify the sample `mc-crusher` parameters. From inspection of the `mc-crusher` source, we expect the three benchmarks to operate on generally disjoint data, and therefore do not anticipate any direct connection between the execution of one and the output of the next. However, it is difficult to predict the presence or magnitude of *indirect* ordering artifacts, where the side effects of previous computation might influence the efficiency of subsequent operations, which is what our analysis aims to measure.

Table 3 presents the results we obtained by running OrderSage with `memcached` version 1.5.22, with 50 fixed and 50 random runs, each including the three tests described.

Overall, we conclude that the order of trials within a run *does* affect the measurements obtained for the `mc-crusher` environment under test, at the 95% significance level. This coincides with the `get_hits`'s median performance changing by over 5% based on whether a fixed-order or random-order experiment design is used.

7.2 NPBench & NPB

NPBench is a "a set of NumPy code samples representing a large variety of HPC applications" [43]. The authors use it to test a variety of Python HPC frameworks and compilers that aim to accelerate NumPy code; they also expect the results to be useful to end-users of such frameworks. NAS Parallel Benchmarks (NPB) is an open source benchmarking suite which includes "a small set of programs designed to help evaluate the performance of parallel supercomputers." [27] We select two tests from NPBench that exercise operations used in data analytics and machine learning: sparse matrix-vector multiplication (`SPMV`) and the normalized exponential function (`softmax`) used in neural networks. In addition, we select integer sort (`IS`) from NPB, which is used to benchmark random access memory. SPMV and softmax are generally CPU-bound, while IS generally has its performance limited by memory speed. Using OrderSage, we did 100 runs in each of fixed and random orders. We set problem sizes to large enough values to get meaningful results on CloudLab machines: flag L in NBbench is expected to take about 1000ms to run whereas class D in NPB is the largest test problem for IS, and the median runtime was 36 seconds.

Table 4: Test results for the NPBench & NPB experiment. Columns are as described for Table 3.

Test	KW p-value	KW test	$\Delta_{\%}$	CI case
`IS`	0.83	○	0.00	2
`SPMV`	0.69	○	−0.60	2
`softmax`	0.03	•	0.46	3

The results from these experiments are in Table 4. `IS` and `SPMV` show no order-dependence. While `softmax` does show a statistically-significant change in distribution when run in a random order, the effect size of 0.46% is small enough that it is unlikely to make a difference in practice: these three tests can be safely run in any order. This demonstrates the need to look at effect sizes as well as statistical significance: a positive result from the Kruskal-Wallis test does not, by itself, guarantee that the effect is large enough to matter.

7.3 uFS Paper Artifact Reproduction

Our final case study looks at the uFS filesystem presented at SOSP 2021 [19]. This paper submitted an artifact and was awarded the Available, Functional, and Reproduced badges; it was part of our survey in Section 2. uFS is a user-level filesystem "semi-microkernel" [19] that claims good base performance and better scalability than the ext4 filesystem in the Linux kernel. This is demonstrated with benchmarks at various scales and under various threading conditions. Using OrderSage, we find that some experiments run for this paper *are* order-dependent with large effects (up to 17%), though not large enough to change the conclusions of the paper.

The evaluation scripts supplied with the artifact run multiple benchmarks, of which we selected the Microbenchmarks with single-threaded uFS and ext4 (both without journaling.) Their scripts run all 32 workloads in sequence; we modified

Table 5: Test results for the uFS experiment. In the original paper, `ufs` results are compared with corresponding `ext4nj` experiments. Columns are as described for Table 3.

Test	KW p-value	KW test	$\Delta_{\%}$	CI case
`ufs.ADSS`	0.028	•	16.8%	2
`ext4nj.ADSS`	0.364	○	-4.0%	2
`ufs.ADPS`	0.013	•	6.7%	2
`ext4nj.ADPS`	0.406	○	4.7%	2
`ufs.RDSR`	0.112	○	0.2%	2
`ext4nj.RDSR`	0.406	○	-0.8%	2
`ufs.RMS`	0.940	○	0.8%	2
`ext4nj.RMS`	0.650	○	0.1%	2
`ufs.LsMS`	0.650	○	-0.6%	2
`ext4nj.LsMS`	0.940	○	0.0%	2
`ufs.RMP`	0.545	○	0.0%	2
`ext4nj.RMP`	0.545	○	0.3%	2
`ufs.CMP`	0.496	○	-0.2%	2
`ext4nj.CMP`	0.256	○	-0.1%	2
`ufs.LsMP`	0.151	○	3.6%	2
`ext4nj.LsMP`	0.406	○	0.1%	2
`ufs.CMS`	0.019	•	-1.3%	2
`ext4nj.CMS`	0.705	○	0.3%	2
`ufs.RDPR`	0.112	○	0.2%	2
`ext4nj.RDPR`	0.226	○	1.9%	2

them to run one workload at a time as individual tests. We use the leftmost data point for evaluation as described in the paper's Section 4.2 and Figure 5a—these are used to evaluate the claim that uFS performs as well as or better than ext4 under baseline, single-threaded conditions. We used OrderSage and a `c6525-100g` node in CloudLab (which has a dedicated NVMe drive as does the original authors' machine) to run these tests in fixed and random orders (10 times each).

Our results (Table 5) show that order does not matter to most tests, but it does matter to three: `ufs.ADSS`, `ext4nj.ADPS`, and `ufs.CMS`, with the `ufs.ADSS` test changing the most: in the fixed order, its median is 119K with a tight CI of $[117K, 120K]$. In random order, its median drops by 16.8% with a much wider CI of $[74K, 121K]$. The conclusion from the uFS paper still holds: the random-order `ufs.ADSS` median of $98K$ is still greater than the $41K$ random-order result for baseline system it is compared to, `ext4nj.ADSS`, and the CIs do not overlap. This effect may be due to hardware differences: the original uFS paper was evaluated on an NMVe drive using Intel Optane memory, while the drive we used on CloudLab has traditional flash memory. As a result, latencies and flush strategies differ between the environments. However, this demonstrates the necessity of avoiding the ordering trap, as such order-dependent results are probably "hidden" in many published results, and likely indicate system effects that the authors may not be fully aware of.

8 Related Work

There is much scientific literature focused on experimental design and analysis of computer systems performance experiments [12, 16, 18, 32, 34]. Among recent work in related areas are studies of presentation flaws specific to performance results [11] and analysis of performance variability in computer systems [22]. In a separate but relevant context, some research and development efforts are focused on *testbeds*, i.e. computer infrastructure, designed for reproducible experiments [28, 42], and how they can facilitate trustworthy experimental evaluations. Studies of computer benchmarking [10, 15] consider both the nuances of benchmark design and interpretation of results. However, the aforementioned sources do not help conclusively answer the question: "Does the order of tests matter, and if so, how much?" Our study aims to bridge this gap.

One recent study related to our work focuses on repeatable experiments in highly variable cloud environments [1]. The authors study the following designs: 1) Single Trial, 2) Multiple Consecutive Trials, 3) Multiple Interleaved Trials (MIT), and 4) Randomized Multiple Interleaved Trials (RMIT). Another study of the RMIT execution plan led to the development of WPBench, a web serving benchmark suite that bundles a set of micro and application benchmarks [35]. The fixed and random orders we study correspond to MIT and RMIT, respectively. While those studies argue for using RMIT, our investigation extends previous work with a large-scale evaluation of both approaches and shows where the differences between the two are most significant. We also consider environments without "background noise" from other tenants.

The idea of turning a proposed methodology into a reusable tool was inspired by the recent work on Lancet, a self-correcting tool for latency analysis [14]. TraceSplitter [33] applies an analogous statistical approach to traffic traces. Similarly, Hyperfine [29] facilitates many tasks involved in the benchmarking process and common subsequent analyses. In turn, experimenters can focus more on creating interesting experiments with increased confidence that their conclusions are unbiased by factors such as test ordering. We implement OrderSage with this vision in mind and describe in this paper the results it collects in several use cases.

Another related study considers performance change-points [6]. The data collection in our work is similar to the process described in that paper. However, rather than characterizing temporal patterns broadly, we focus on the order-related effects and the methodology for studying them.

9 Conclusion and Future Work

The order in which tests are run is a significant, but often neglected, part of experiment design—as shown in our survey, it is rarely mentioned in papers, and the artifacts that support them show little sign of being designed with ordering in mind. Our findings show that order can indeed make a difference: sometimes quite a large one. Systems experimenters should take this into account in their experiment designs, and test for order dependence when feasible. The response to discovered order-dependence will vary depending on the system, the experiment, and its goals. In some cases, there may be aspects

of the system under test, test environment, or test procedure that need to be "fixed" to make runs more consistent and less dependent on order. In other cases, some amount of variability is simply to be expected, and experiments should be run in several randomized orders to avoid systematically biasing results with a single order. Finally, in some cases, it may be that a "clean" environment is not the most realistic one in which to run the experiment, and more effort needs to be taken to get the environment into a suitably realistic state.

Our work thus far has left out of scope a deep analysis of *why* order matters. This would be an interesting subject for follow-up research, and we expect that the reasons will be as varied as the tests that are run and the environments they are run in. One way to do such an investigation would be to analyze which tests *cause* changes in the following trials, and which ones see the largest *effects*. We hope that our open dataset and tool will help to enable such explorations.

References

[1] Ali Abedi and Tim Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 287–292, 2017.

[2] Damiano Carra and Pietro Michiardi. Memory partitioning in memcached: An experimental performance analysis. In *Proceedings of the IEEE International Conference on Communications (ICC)*, June 2014.

[3] A. Cohen, F. Finkelstein, A. Mendelson, R. Ronen, and D. Rudoy. On estimating optimal performance of CPU dynamic thermal management. *IEEE Computer Architecture Letters*, 2(1), 2003.

[4] Barry H. Cohen. *Explaining Psychological Statistics*. John Wiley & Sons, 2008.

[5] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, July 2019.

[6] Dmitry Duplyakin, Alexandru Uta, Aleksander Maricq, and Robert Ricci. In datacenter performance, the only constant is change. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 370–379. IEEE, 2020.

[7] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2021.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[9] Yosef Hochberg and Ajit C Tamhane. *Multiple Comparison Procedures*. Wiley, 1987.

[10] Roger W Hockney. *The science of Computer Benchmarking*. SIAM, 1996.

[11] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, page 73. ACM, 2015.

[12] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley- Interscience, April 1991.

[13] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008.

[14] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting latency measuring tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 881–896, 2019.

[15] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking*. Springer, 2020.

[16] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowskiu. *Systems Benchmarking For Scientists and Engineers*. Springer Nature Switzerland AG, Cham, Switzerland, 2020.

[17] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, 2011.

[18] Dennis KJ Lin, Timothy W Simpson, and Wei Chen. Sampling strategies for computer experiments: design and analysis. *International Journal of Reliability and applications*, 2(3):209–240, 2001.

[19] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the*

ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[21] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: bringing legacy code to byte-addressable persistent memory. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, July 2017.

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

[23] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[24] Memcached community. mc-crusher. https://github.com/memcached/mc-crusher, 2011–2020.

[25] R. G. Miller. *Simultaneous Statistical Inference (2nd Ed.)*. Springer Verlag, New York, 1981.

[26] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[27] National Aeronautics and Space Administration, Advanced Supercomputing Division. Nasa parallel benchmarks (npb). https://www.nas.nasa.gov/software/npb.html, 2022.

[28] Lucas Nussbaum. Testbeds support for reproducible research. In *Proceedings of the Reproducibility Workshop*, Reproducibility '17, pages 24–26, New York, NY, USA, 2017. ACM.

[29] David Peter. Hyperfine: a command-line benchmarking tool. https://github.com/sharkdp/hyperfine, 2023.

[30] Alex W. Reece. Achieving maximum memory bandwidth. http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html, May 18 2013.

[31] Alex W. Reece. Memory bandwidth demo. https://github.com/awreece/memory-bandwidth-demo, May 19 2013.

[32] Jerome Sacks, William J Welch, Toby J Mitchell, and Henry P Wynn. Design and analysis of computer experiments. *Statistical science*, 4(4):409–423, 1989.

[33] Sultan Mahmud Sajal, Rubaba Hasan, Timothy Zhu, Bhuvan Urgaonkar, and Siddhartha Sen. Tracesplitter: A new paradigm for downscaling traces. In *16th European Conference on Computer Systems (EuroSys)*, April 2021.

[34] Thomas J Santner, Brian J Williams, William I Notz, and Brain J Williams. *The Design and Analysis of Computer Experiments*, volume 1. Springer, 2003.

[35] Joel Scheuner and Philipp Leitner. A cloud benchmark suite combining micro and applications benchmarks. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 161–166, 2018.

[36] David J Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, Boca Raton, Florida, 2000.

[37] The CloudLab Team. CloudLab hardware. https://www.cloudlab.us/hardware.php, 2018.

[38] The CloudLab Team. Describing a profile with python and geni-lib. http://docs.cloudlab.us/geni-lib.html, August 2018.

[39] The CloudLab Team. Collection script for cloudlab benchmarks. https://gitlab.flux.utah.edu/emulab/cloudlab-orchestration, 2021. Accessed: 2021-10-17.

[40] Maciej Tomczak and Ewa Tomczak. The need to report effect size estimates revisited. an overview of some recommended measures of effect size. *Trends in Sports Sciences*, 1(21):19–25, 2014.

[41] E. Weigle and Wu chun Feng. A comparison of TCP automatic tuning techniques for distributed computing. In *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, 2002.

[42] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In

Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI). USENIX, December 2002.

[43] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. Npbench: A benchmarking suite for high-performance numpy. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, New York, NY, USA, 2021. Association for Computing Machinery.

Appendix A Using OrderSage

Using OrderSage is straightforward, and requires little beyond that used in a typical experiment.

- **Experiment Environment:** Users must have a controller node and at least one worker node that has remote-access capabilities. The controller is separate from the worker so that the latter can be rebooted as part of the reset procedure.
- **Experiment Repository:** The tests to run and their associated scripts are stored in a `git` repository created by the user, which makes them natively version-controlled. In addition to the system(s) under evaluation, the repository contains the following:
  - **Test Configuration Script:** Called during the initialization phase by the controller, this script prints a list of commands to `stdout`. The commands will be executed in-order for the fixed runs and shuffled for the random runs. Each command represents a single test and all commands must be unique. It is up to users to implement this script as they wish as long as these requirements are met. In the simplest case, it can be a series of print statements of varying test commands or it can be more complex and include methods to iterate through complex sets of parameters, producing a command for each one.
  - **Initialization Script:** The controller calls an initialization script to ready all workers for experimentation as defined by the user. This script can install packages, set machine states, etc. Its only requirement is that it creates a "results" directory in a location on the worker.
- **Configuration File:** To run OrderSage, the user creates a configuration file. This configuration contains the URL of the experiment repository, the location of the configuration and initialization scripts within the repository, and other parameters. These parameters include paths to results files, the number of runs, etc. If the set of worker node(s) is pre-allocated, the `workers` parameter of this file must contain a list of all worker node hostnames.
- **Define Reset Protocol:** Our default implementation of OrderSage calls `reset()`, which is implemented to reboot the worker node(s) and reconnect between runs. However, if users prefer a different reset procedure, they can override this method.
- **Results:** Results are collected in a single text file on each worker node. Each test run (trial) must provide a single, floating-point number on a new line of the file. It is important that this results file is presented in-order (i.e., the first trial produces the first number and the n^{th} trial produces the n^{th} number). In total, the number of lines in the result file must equal the *number of tests* $\times$ *the number of runs* $\times$ 2 (for fixed and random runs).

Once the aforementioned configuration is complete, a user can run OrderSage by executing the following command:

```
# python controller.py
```

The artifact with the code and data we released, https://github.com/ordersage/paper-artifact, has more information on running OrderSage and reproducing the results presented in this paper.

AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems

Haoran Qiu[1] Weichao Mao[1] Chen Wang[2] Hubertus Franke[2] Alaa Youssef[2]
Zbigniew T. Kalbarczyk[1] Tamer Başar[1] Ravishankar K. Iyer[1]

[1]*University of Illinois at Urbana-Champaign* [2]*IBM Research*

Abstract

Workload autoscaling is widely used in public and private cloud systems to maintain stable service performance and save resources. However, it remains challenging to set the optimal resource limits and dynamically scale each workload at runtime. Reinforcement learning (RL) has recently been proposed and applied in various systems tasks, including resource management. In this paper, we first characterize the state-of-the-art RL approaches for workload autoscaling in a public cloud and point out that there is still a large gap in taking the RL advances to production systems. We then propose AWARE, an extensible framework for deploying and managing RL-based agents in production systems. AWARE leverages meta-learning and bootstrapping to (a) automatically and quickly adapt to different workloads, and (b) provide safe and robust RL exploration. AWARE provides a common OpenAI Gym-like RL interface to agent developers for easy integration with different systems tasks. We illustrate the use of AWARE in the case of workload autoscaling. Our experiments show that AWARE adapts a learned autoscaling policy to new workloads 5.5× faster than the existing transfer-learning-based approach and provides stable online policy-serving performance with less than 3.6% reward degradation. With bootstrapping, AWARE helps achieve 47.5% and 39.2% higher CPU and memory utilization while reducing SLO violations by a factor of 16.9× during policy training.

1 Introduction

Motivation. Reinforcement learning (RL) has become an active area in machine learning research and is widely used in various systems tasks (e.g., resource scaling [23,47–49,59,62], power management [58, 64], job scheduling [4, 5, 32, 33, 35, 63], video streaming [34, 60], and congestion control [22, 28, 31, 56, 60]). As a viable alternative to human-generated heuristics, RL automates the repetitive process of heuristics tuning and testing by enabling an *agent* to learn the optimal *policy* directly from interaction with the *environment*.

One example is workload resource autoscaling for meeting application service-level objectives (SLOs) while achieving high resource utilization efficiency [8, 30, 47, 49, 50, 59]. Traditional rule-based approaches [2, 3, 6, 25] configure static upper and lower thresholds offline for certain system metrics (e.g., CPU or memory utilization) or application metrics (e.g., request arrival rate, throughput, or end-to-end latency) so that resources can be scaled accordingly when the measured metrics go above or below the thresholds. Tuning and testing of fine-grained thresholds require significant application/system-specific domain knowledge from experts to achieve optimal resource allocation. Further, repeated parameter tuning for each workload can be labor-intensive, especially for microservice-like applications in large-scale production systems. As different types of services may use different amounts of resources (e.g., CPU and memory) and are sensitive to different kinds of interference and workload spikes, a customized threshold has to be set for a service differently.

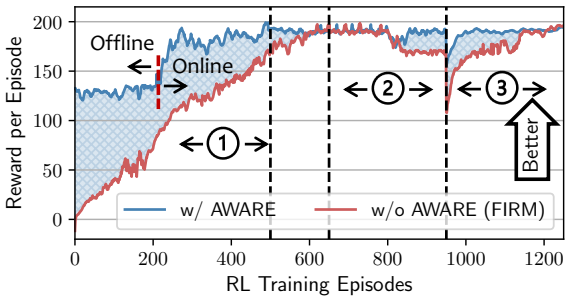


Figure 1: RL agent performance when managed by AWARE compared to the baseline (FIRM [47]). Stages ①, ②, and ③ demonstrate the benefit of RL bootstrapping, incremental retraining, and fast adaptation, respectively.

RL, on the other hand, is well-suited for learning optimal policies, as it models a systems task (e.g., workload autoscaling) as a sequential decision-making problem and provides a tight feedback loop for exploring the state-action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules) [32, 48]. Integrating RL with those complex systems management tasks in production systems can (a) make full use of the abundant monitoring data on applications and the infrastructure, and (b) automate the process of developing optimal policies while freeing operators from manual workload profiling and parameter tuning/testing. For example, FIRM [47]'s RL agent learns an optimal workload autoscaling policy that adapts to specific application workloads with online telemetry data that alleviates the need for handcrafted heuristics (see §2.2 for details).

Challenges. However, even as RL is starting to show its strength in the systems and networking domains [4, 5, 22, 23, 28, 31–35, 47–49, 56, 58–60, 62–64], there is still a large

gap in directly applying RL advances to real-world production systems due to a series of assumptions that are rarely satisfied in practice. First, a learned RL policy is workload-specific and infrastructure-specific. Retraining is needed to adapt to a new workload or underlying infrastructure in heterogeneous and dynamically evolving (possibly multi-cloud) datacenters [18,37,53,54,58]. For instance: (a) In SLO-driven resource management, application performance and utilization differ significantly among heterogeneous workloads [47]. Fig. 1 stage ③ shows that FIRM's trained RL policy suffers performance degradation and requires substantial retraining. (b) In power management, diverse power consumption and workload sensitivity to core/uncore frequency require separate training of RL policies [58]. (c) In video streaming and network congestion control, different sets of traces have diverse payload characteristics and network environments [60] (e.g., dynamic link bandwidth, delay, and loss rate). Even with transfer learning (TL) [47], nontrivial retraining is needed to adapt to new workloads and environment shifts in each problem domain, which is a critical problem in making RL practical in production. Further, TL requires fine-grained environment clustering to identify the most appropriate model to transfer from, and requires saving one model per cluster.

Second, for the same application and environment, there could be slight changes (e.g., patches and rolling updates), unusual workload patterns not seen before (e.g., due to migration rollout), or traffic jitters. Without timely retraining, the online policy-serving performance of the RL agent fluctuates and leads to undesired degradation (as shown in Fig. 1 Stage ②). It is crucial to ensure robust online performance in case of environment or model uncertainty [40,46].

Third, RL training is through trial and error [32,35,47], so worse-than-baseline or suboptimal decisions can be generated, especially at an early stage of training (as shown in Fig. 1 Stage ①). Direct training in the production system leads to suboptimal performance and undesired SLO violations, while training in a simulator and then transitioning to the production system face the problem of poor generalization [61].

A framework that can bring the RL advances to production systems is needed so that (a) the RL model can be trained in a safe and robust manner, (b) the learned RL policy can be adapted to new workloads and altered environments seamlessly without significant retraining, and (c) the online RL model policy-serving performance can be kept stable.

Our Work. We first performed a characterization study of RL in production systems in the task of workload autoscaling. The study focused on the impact of workload change and environment shift regarding (a) RL agent performance degradation or variation and (b) retraining cost. To facilitate the deployment and operation of RL agents in the systems management tasks of a production cloud environment, we introduce an RL model-serving and management framework. As a general framework to support a variety of RL agents for systems management tasks, it can be used by system operators to develop RL-based agents that can be quickly adapted to new environments and achieve stable online policy-serving performance with continuous monitoring and safe bootstrapping (as shown in Fig. 1). In the end, system operators can benefit from RL automation of systems management tasks.

- To achieve **fast model adaptation** in each domain or systems task, we leverage meta-learning [39] to model the RL agent as a *base-learner* and create a *meta-learner* for learning to generalize and adapt to new applications and environment shifts. The base-learner discovers policies that generalize across workload variations and intra-environment dynamicity for an <application, environment> pair, while the meta-learner generalizes across <application, environment> pairs to address inter-environment dynamicity and application heterogeneity. We designed a novel framework that allows the meta-learner to learn to generate an *embedding* [38, 39, 57] that projects the application- and system-specific data to a vector space. On this projected vector space, workloads with similar characteristics are projected to closer locations, while those with quite different characteristics are projected to locations far from each other. The embedding is generated by encoding a set of *episodes* from the RL agent's exploration of the environment. Since each episode records a step-by-step interaction of the RL agent with the environment, the time-series episodes naturally encode spatial and temporal characteristics. In the task of workload autoscaling, spatial characteristics correspond to the workload's performance sensitivity to different resource allocations, and temporal characteristics correspond to the time-varying load patterns. The generated embedding is then fed as input to the base-learner to adapt to the application and environment shift (from the environments with similar characteristics). With the embedding, fewer retraining iterations are needed for new, previously unseen workloads.
- To achieve **stable online RL policy-serving performance**, we leverage continuous monitoring, and designed a retraining detection and trigger mechanism. An RL agent observes a *state*, performs an *action*, and gets a *reward* at every step in an episode. The time series of states, actions, and rewards in an episode form a *trajectory*. RL trajectories are collected and stored in a time-series database. The most recent rewards are used to calculate the average reward and variation for comparison against user-specified targets. Continuous monitoring ensures that RL model retraining can be triggered or stopped timely so that the RL policy can seamlessly adapt to any environment jitters. We intercept the RL model update logic to enable the switch between RL policy serving and retraining.
- To achieve **safe RL exploration**, we designed an RL bootstrapping module that combines offline and online training. The agent starts with offline training, and a traditional heuristics-based controller (e.g., the Kubernetes Horizontal Pod Autoscaler (HPA) [25] and the Vertical Pod Autoscaler (VPA) [15] in the case of workload autoscaling), is used as the navigator for (online) exploration of the state and action space in the environment. After the RL model is trained to the

same level as the heuristics-based controller by comparing rewards, the agent continues to be trained online.

We have demonstrated the proposed framework in the task of workload autoscaling on Kubernetes by implementing AWARE (i.e., **A**utomate **W**orkload **A**utoscaling with **RE**inforcement learning). Each RL agent manages a Kubernetes Deployment and configures resources automatically, adjusting both the number of replicas (horizontal scaling) and the CPU/memory limits (vertical scaling) to maintain workload service-level objectives (SLOs) and achieve high resource utilization. To integrate RL agents with Kubernetes, we designed and implemented a multidimensional Pod autoscaler (MPA) system. MPA provides system support for RL-based controllers and translates RL outputs into multidimensional autoscaling actions in a holistic manner by (a) providing an API for RL agents to execute horizontal and vertical scaling decisions on Pod CPU and memory limits, (b) combining vertical and horizontal scaling actions in a single CRD object [24], and (c) providing a user interface for user-defined objective functions for multidimensional autoscaling.

Results. We present a detailed experimental evaluation of AWARE, demonstrating that AWARE significantly improves the practicality of applying RL in production cloud systems (for workload autoscaling). We first show that the adaptation process of a learned autoscaling policy to new workloads with meta-learning is 5.5× faster than the existing transfer-learning-based approach (§5.2), and then demonstrate that AWARE provides stable online policy-serving performance with less than 3.6% reward degradation (§5.3). AWARE's bootstrapping mechanism helps achieve 47.5% and 39.2% higher CPU and memory utilization while reducing SLO violations by a factor of 16.9× during training (§5.4).

Contributions. In summary, our main contributions are:

- A characterization of RL-based production workload autoscaling and the challenges involved in applying RL in production cloud systems (§2.3).
- The design of a novel meta-learning-based framework for fast RL model adaptation in workload autoscaling (§3.2).
- The design of an RL retraining management and bootstrapping mechanism for stable policy-serving performance (§3.3) and robust RL environment exploration (§3.4).
- An implementation of the proposed framework in the task of workload autoscaling with MPA, which enables integration of RL agents with Kubernetes (§4.1).
- A detailed evaluation of AWARE that demonstrates substantial improvements through meta-learning and RL lifecycle management while maintaining workload SLOs and resource utilization (§5).

2 Background & Characterization

2.1 Reinforcement Learning

In reinforcement learning (RL), an *agent* interacts with an *environment* modeled as a discrete-time Markov decision process (MDP) (as shown in Fig. 2). At time step t, the agent

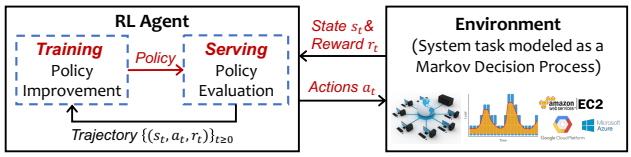


Figure 2: An RL agent interacting with an environment modeled as a systems task (e.g., workload autoscaling or congestion control) in the form of a Markov decision process (MDP).

perceives a *state* $s_t \in S$ of the environment and takes an *action* $a_t \in A$. The agent receives a *reward* $r_t \in \mathbb{R}$ as feedback on how good the decision is, and at the next time step $t+1$, the environment transitions to a new state s_{t+1}. The whole sequence of transitions $\{(s_t, a_t, r_t)\}_{0\leq t\leq T}$ is called a *trajectory* or *episode* of length T. The agent's goal is to learn a *policy* π_θ[1] that maximizes the expected cumulative rewards in the future, i.e., $\mathbb{E}[\sum_{t=0}^{T} \gamma^t \cdot r_t]$, where the discount factor $\gamma \in (0,1)$ progressively de-emphasizes future rewards. RL consists of a *policy-training* stage and a *policy-serving* stage [41]. At the policy-training stage, the agent (using an initialized policy) starts with no knowledge about the task and learns by reinforcement and directly interacting with the environment. At the policy-serving stage, the trained policy is used to generate an action based on the current state of the environment, and model parameters are no longer being updated.

2.2 Workload Autoscaling with RL

Because of the sequential nature of the decision-making process, RL is well-suited for learning resource management policies, as it provides a tight feedback loop for exploring the state-action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules) [32, 48]. In addition, since the decisions made for workloads are highly repetitive, an abundance of data is generated to train such RL algorithms even with deep neural networks[2]. By directly learning from the actual workload and operating conditions to understand how the allocation of resources affects application performance, the RL agent can optimize for a specific workload and adapt to varying conditions in the learning environment. RL [23, 47–49, 59, 62] has been shown to automate resource management and outperform heuristics-based approaches in terms of meeting workload SLOs and achieving higher resource utilization.

Specifically, we adopted the design and took the open-source implementation of an RL-based workload autoscaler from FIRM [47], which is the state-of-the-art RL-based autoscaling solution, to the best of our knowledge. FIRM uses an actor-critic RL algorithm called DDPG [29].

The RL agent monitors the system- and application-specific measurements and learns how to scale the allocated resources vertically and horizontally. Table 1 shows the model's state

[1] A policy π_θ maps the state space S to the action space A and is usually represented by neural networks (with parameters denoted by θ).

[2] Deep neural networks can express complex system-application environment dynamics and decision-making policies but are data-hungry.

Table 1: State-action space of the RL agent.

State Space (s_t)
Resource Limits (CPU, RAM), Resource Utilization (CPU, Memory, I/O, Network), SLO Preservation Ratio (Latency, Throughput), Observed Load Changes
Action Space (a_t)
Resource Limits (CPU, RAM), Number of Replicas

Table 2: RL training hyperparameters.

Parameter	Value
# Time Steps per Episode	100×64 mini-batches
Replay Buffer Size	10^6
Learning Rate	Actor (3×10^{-4}), Critic (3×10^{-3})
Discount Factor	0.99
Soft Update Coefficient	3×10^{-3}
Random Noise	μ (0), σ (0.2)
Exploration Factor	ε (1.0), ε-decay (10^{-6})

and action spaces. The goal is to achieve high resource utilization (RU) while maintaining application SLOs (if there are any). SLO preservation (SP) is defined as the ratio between the SLO metric and the measured metric. If no SLO is defined for the workload (e.g., best-effort jobs) or the measured metric is smaller than the SLO metric, $SP = 1$. An SLO metric can be either request serving latency (e.g., the 99th percentile of the requests are completed in 100ms) or throughput (e.g., request processing rate is no less than 100/s). The reward function is then defined the same as in FIRM [47], $r_t = \alpha \cdot SP_t \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_{i \in \mathcal{R}} RU_i$, where $\mathcal{R}$ is the set of resources. Table 2 lists the hyperparameters tuned for better performance in the experiments. The RL algorithm is trained in an episodic setting. In each episode, the agent manages the autoscaling of the application workload for a fixed period of time (100 RL time steps in our experiments).

2.3 Characterization of RL in Production

In the characterization study of FIRM for workload autoscaling, we selected 16 representative production cloud workloads based on a survey of 89 industry use cases of serverless computing applications [11], as serverless workloads are highly dynamic (and thus require autoscaling) and rely on the provider to manage the resources. The selected production workloads include CPU-intensive tasks (e.g., floating-point number computation), image manipulation, text processing, data compression, web serving, ML model serving, and I/O services (e.g., read, write, and streaming). Next, we deployed the selected workloads as Deployments in a five-node Kubernetes cluster in a public cloud and ran an RL-based multi-dimensional autoscaler (i.e., a FIRM agent) with each workload. All nodes run Ubuntu 18.04 with four cores, 16 GB memory, and a 200 GB disk. For RL agent training and inference, we used real-world datacenter traces [65] released by Microsoft Azure, collected over two weeks in 2021.

We next present the key insights from the characterization study in the order of adaptation, online policy-serving, and early-stage of RL training.

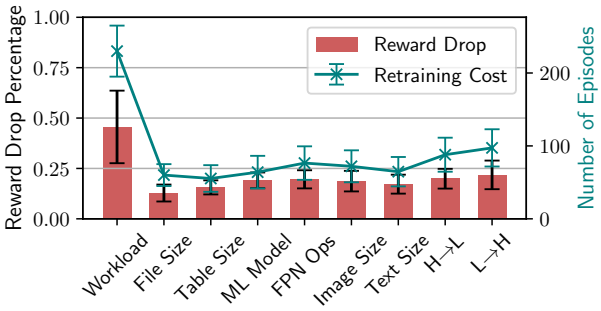


Figure 3: Retraining cost of RL models.

Insight 1: Adaptation Retraining Cost. To study the retraining cost of adapting a trained RL policy to new application workloads, we selected each application from the workload pool, trained an RL agent for the application until convergence, and retrained the learned RL policy to serve all the other different applications. We then measured the reward drop after the workload changed and the number of episodes each agent took to retrain to convergence. As shown in Fig. 3 (column 1), we observed a 45.6% average per-episode reward drop percentage when the workload had been changed, and retraining to convergence required around 230 episodes (with the model parameter transfer learning used in FIRM [47]).

Insight 2: Online Policy-serving Performance Jitters. We introduced seven scenarios to explore the performance instability of RL-based workload autoscaling agents when facing application or service payload size changes and load pattern changes. For I/O services to a backend file system (e.g., AWS S3) and the compression/decompression services, the size of files being read, written, or streaming was changed from [128 KB, 256 KB, 384 KB] to [512 KB, 768 KB, 1024 KB]. For database services, the size of the table being scanned was changed from 1024 items to 10240 items. For floating-point number calculation, the number of operations was changed from 10^8 to 20^8. For image manipulations, the dimension was changed from 40×40 to 160×160. For text processing, the JSON file size was changed from [250 B, 500 B, 1 KB] to [2 KB, 3 KB, 5 KB]. For ML model serving, we changed the matrix multiplication dimension from 50 to 150. For load pattern changes, we divided the Azure workload traces into two parts, one half with a higher daily load ($> 10^5$ per day) and the other half with a lower daily load ($\leq 10^5$ per day).

Fig. 3 (columns 2–9) shows the per-episode reward drop percentage and the retraining cost of each scenario. File size changes led to the lowest 12.8% reward drop and around 70 episodes of retraining. We attribute this to I/O-intensive workloads' relatively low sensitivity to CPU/memory allocation, compared to compute- or memory-intensive workloads. Other payload-related changes (i.e., table size, ML model, floating-point number operations, image dimension, and text size) resulted in a 15.6–19.9% reward drop. Load changes from high request arrival rates to low arrival rates (i.e., H→L

Table 3: Workload performance and utilization efficiency deficit (i.e., the relative difference compared to the rule-based approach) in early-stage RL model training.

RL Episodes	EP 1–100	EP 101–200	EP 201–300	EP 301–400
CPU Util	-32.3% ± 14%	-42.9% ± 15%	-22.1% ± 12%	-10.0% ± 6%
Memory Util	-28.8% ± 11%	-30.5% ± 10%	-26.5% ± 8%	-7.8% ± 2%
SLO Violations	56.1 ± 14 ×	22.2 ± 7 ×	12.7 ± 5 ×	10.1 ± 3 ×

in Fig. 3) and from low rates to high rates (i.e., L→H) resulted in 19.9% and 21.8% reward drops, which required around 98 and 107 episodes of model retraining, respectively.

Insight 3: Cost of Early-stage RL Training. As mentioned in §2.2, RL training proceeds in episodes. When the initialized RL agent starts to learn the optimal policy, especially at an early stage of policy training, the policy might be worse than the baseline heuristics-based approach or even produce undesired actions, such as an oscillating scaling up and down behavior. This is primarily due to the exploration of the state-action space and RL agent learning through trial and error. To study what is lost during policy training, we compared workloads managed by RL agents with the same workloads managed by the rule-based autoscaling approach (i.e., HPA and VPA). We define the early stage of RL training to be the training process from the beginning to the episode at which the RL agent starts to get better than the rule-based approach (which is around 400 episodes in our experiments) because we are interested in the loss due to RL training compared to non-RL-based approaches. We then divided the 400 episodes (in the early training stage) into four segments. For each segment, we calculated the accumulated utilization deficit and SLO violations of the application workloads controlled by the RL agents; the results are shown in Table 3. The relative difference in utilization or SLO performance is based on the comparison between the RL agent and the rule-based approach when used to control the same application workloads with the same set of traces.

Results show that RL policies necessarily lead to poor decisions in the early stages of training. In the first 100 episodes, the RL agents inevitably caused more SLO violations than in the other segments (56.1× more than the rule-based approach, which had five SLO violations per 100 episodes). We observe that most SLO violations were due to the under-provisioning of resources, so the CPU and memory utilization deficits (32.3% and 28.8% lower, respectively, than for the rule-based approach) were smaller than those in the later segments. In the last three segments, we observe a utilization deficit (i.e., 10–42.9% lower CPU utilization and 7.8–30.5% lower memory utilization) and more SLO violations (i.e., 10.1–22.7×) compared to the rule-based approach.

Summary and Implications. Workloads running in production cloud systems might be user-facing or high-stakes. *To enjoy the benefit of RL in systems management, the key challenge is to produce fast-adapting, effective, and robust RL-based solutions under the constraints of production cloud systems.* As of now, to the best of our knowledge, there are no systems that can help agent developers address this challenge.

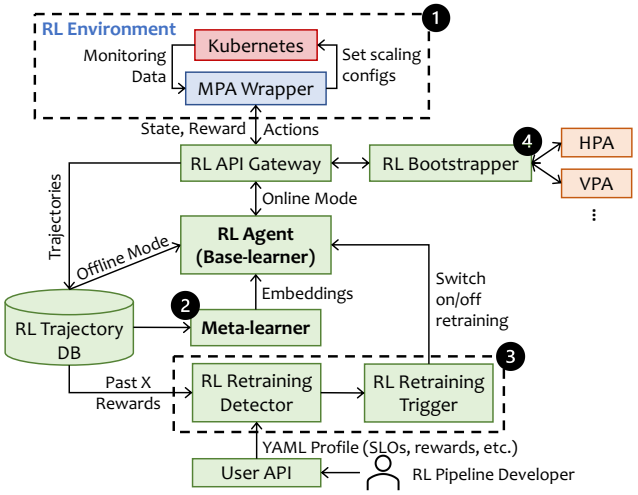


Figure 4: Overview of AWARE.

3 AWARE Design

3.1 Overview

Driven by the insights from §2.3, we describe the design of AWARE, a framework that supports RL agents for multidimensional Pod autoscaling (MPA) of workloads in production Kubernetes systems. AWARE manages the RL agent lifecycle to deliver stable and robust agent performance. Fig. 4 provides an overview of AWARE. We next present a brief summary of each component in this section.

RL Environment. The RL environment (denoted by ❶ in Fig. 4) of AWARE consists of a cluster deployment (e.g., Kubernetes) and an MPA wrapper. The MPA wrapper is designed and implemented as a shim layer that follows an "agent-centric" pattern of request-response interaction advocated by OpenAI Gym [44]. The purpose of the MPA wrapper is to translate measurements and scaling recommendations to and from RL abstractions (i.e., states/rewards and actions), respectively. The communication between the wrapper and the RL agent is through remote procedure calls (RPCs). When the agent steps the environment forward by sending an action to the MPA wrapper through the RPC request, the wrapper translates the received action to vertical and horizontal scaling configurations and applies it to the cluster deployments (e.g., by setting the VPA object [15] and calling the replica re-scaling API). The wrapper gets measurements from the monitoring service in the cluster (e.g., Prometheus [7] in Kubernetes), translates them to RL states and rewards, and sends them back to the agent through the RPC response. The wrapper then waits on the RPC server for the next action request. We describe implementation details in §4.1.

The framework can also be applied to other systems management tasks (e.g., job scheduling or network congestion control) by replacing the RL environment. Decoupling the RL environment (i.e., the environment wrapper) from the rest of the framework and using the standard OpenAI Gym interface

make environment replacement easy [33].

RL API Gateway. The RL API gateway connects the RL agent to the MPA wrapper by sending the RL action in an RPC request and unpacking the state and reward in the RPC response for the RL agent. Each RL trajectory consists of <state, action, reward> transitions in one episode where the RL environment defines the length or the terminating condition of an episode. The trajectories from each RL agent, along with the logical timestamp (i.e., the episode and time step index), are saved to the RL trajectory database.

RL Agent (Base-learner). The RL agent implements the DDPG RL algorithm (as described in FIRM [47]) and interacts with the RL environment to perform policy training or policy serving (i.e., inference). Since the interface between the RL agent and the MPA wrapper follows the OpenAI Gym standard, different advanced RL algorithms can be used to replace the original RL algorithm DDPG.

Meta-learner. To help adapt to new workloads or environment updates within the problem domain, the meta-learner (denoted by ❷) selects RL trajectories from the database and generates an embedding that accurately represents the workload running in the environment. RL trajectories are selected per application, and the criteria are based on the reward associated with each trajectory. The embedding is then fed to the base-learner (i.e., the RL agent) as part of the input. The RL agent leverages the embedding to adapt (fine-tune) its policy by differentiating heterogeneous workloads and environment updates. See §3.2 for more details.

RL Retraining Detector and Trigger. At the end of each episode, the RL retraining detector (denoted by ❸) pulls the recent episode rewards gained by the agent from the trajectory database. The mean and standard deviation of the per-episode rewards are calculated and compared to predefined thresholds for performance and variability assessment. If conditions are met, the RL retraining trigger will intercept the inference or training loop of the RL agent to switch retraining on or off, respectively. See §3.3 for more details.

RL Bootstrapper. The RL bootstrapper (denoted by ❹) determines whether the RL training is online or offline. In the online RL training mode, the RL agent interacts directly with the RL environment. However, offline RL training avoids worse-than-baseline performance or illegal actions in the early stages of RL training, which is desired by production systems. In the offline RL training mode, the RL policy training happens offline based on data collected using a fallback option (i.e., a heuristics-based method), while the RL policy is not used for interacting with the environment. The RL bootstrapper intercepts the request-response path between the RL agent and the RL API gateway and replaces the RL agent with the controller implemented as the fallback option. For instance, in the case of workload autoscaling, the default autoscalers widely used are the traditional rule-based approaches HPA (for horizontal scaling) and VPA (for vertical scaling). Given the states at each time step, corresponding autoscaling actions

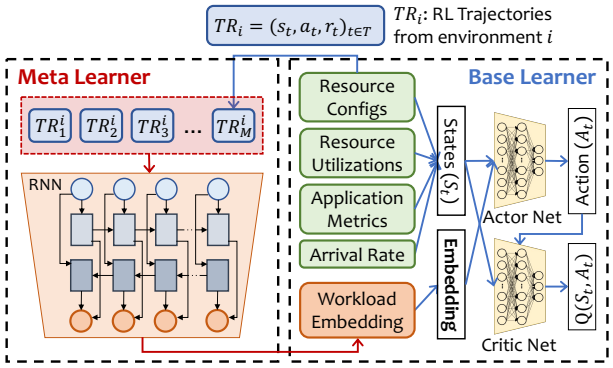


Figure 5: Architecture of meta-learning for RL.

are then generated based on the HPA and VPA algorithms and sent back to the RL API gateway for execution. The trajectories recorded in the RL trajectory database will be used for the offline RL policy training. See §3.4 for more details.

3.2 Meta-learner

Traditional RL-based resource management approaches [23, 47, 48, 59, 62] require the collection of large amounts of training data samples and retraining (even with transfer learning) to adapt to new environments for (a) updated or previously unseen application workloads or (b) constantly evolving cloud infrastructures [18, 37, 53, 58]. Pure RL-based approaches are no longer tenable in such dynamic cloud environments or even in the context of multi-cloud computing [54]. A novel approach that provides fast model adaptation is needed to make RL practical in production cloud systems.

In AWARE, we leverage meta-learning to reduce the retraining overhead and thus adapt quickly to new environments. In essence, the RL agent is treated as the *base-learner* for an individual environment, and a *meta-learner* is designed to generate representative *embeddings* that help differentiate environments. We next give a brief primer on meta-learning and the concept of embeddings before presenting AWARE's meta-learning model and an interpretation of embeddings from a systems perspective.

Meta-learning Primer. Meta-learning is known as learning to learn [26]. A good meta-learning model is capable of adapting well or generalizing to new environments that have never been encountered during training time. The adaptation process, essentially a mini-learning session (with limited exposure to the new environment), happens after the meta-learning model training stage. In the meta-learning model training stage, rather than training the learner on a single environment (with the goal of generalizing to unseen "intra-environment" samples from a similar data distribution), a meta-learner is trained on a distribution of environments, with the goal of learning a strategy that generalizes to unseen environments (i.e., "inter-environment"). Even without any explicit fine-tuning (i.e., with no gradient back-propagation on trainable variables), the meta-learning model autonomously adjusts internal hidden states to learn [12, 20, 39, 43].

Embedding Techniques. Embeddings map variables to low-dimensional vectors in a way that similar variables are close to each other [38, 57]. Embeddings have been widely used in the area of NLP and software engineering (e.g., word or code embeddings) and can also be applied to dense data to create a meaningful similarity metric. In AWARE, embeddings are used to explicitly represent and differentiate environments, and meta-learning enables learning to generate embeddings.

AWARE's Meta-learning Model Design. There are three key components in the design of the meta-learning model:

- A Distribution of MDPs (i.e., RL environments): Each MDP corresponds to one agent to which the base-learner will adapt. During the training of each agent, the meta-learner is exposed to a variety of environments and is trained to adapt to different MDPs. In our case of workload autoscaling, each environment represents a different application workload managed by the base-learner, where workloads can have heterogeneous SLOs, payloads, or architecture.
- A Model with Memory: We use a recurrent neural network (RNN) [17, 52, 55] that maintains a high-dimensional hidden state with nonlinear dynamics to acquire, process, and memorize knowledge about the current environment. In an RNN, hidden layers are recurrently used for computation. Compared to memoryless models such as autoregressive models and feed-forward neural networks, RNNs store information in the hidden states for a long time, so they are effective in capturing both spatial and temporal patterns. We did not explicitly use memory augmentation [51] for our RNN meta-learner because we found that the features of our application workloads are not as high-dimensional as those of computer vision tasks [51], and the RNN hidden states suffice to provide good representations.
- Meta-learning Algorithm: A meta-learning algorithm learns to update the base-learner to optimize for the purpose of adapting quickly to a previously unseen environment [20, 39]. Our novel approach uses an ordinary gradient descent update of RNN with a hidden state reset at a switch of MDPs. As training proceeds, the algorithm learns how to generate an embedding to best represent the environment and differentiate one environment from another.

Integration between Meta-learner and Base-learner. The base-learner discovers a rule that generalizes across data points for an <application, environment> pair, while the meta-learner generalizes across <application, environment> pairs. Fig. 5 illustrates the interaction between the meta-learner (❷ in Fig. 4) and the base-learner. Suppose that each data point used in the training and inference of the RL agent (i.e., a base-learner) with the <application, environment> pair i is $\{(s_t, a_t, r_t)\}_{0 \le t \le T}$, i.e., one RL trajectory TR_i from the environment i; then, each data point in the meta-learner is a bundle of M trajectories from the same environment, i.e., $[TR_1^i, TR_2^i, \ldots, TR_M^i]$. These episodes contain characteristics of the ongoing task that can be used to abstract some specific information about the environment (through <state, action, reward> transition sequences). The meta-learner uses a bidirectional RNN [52] to generate an embedding given a sequence of RL trajectories from the same environment (same base-learner). Unidirectional RNN has the limitation that it processes inputs in strict temporal order, so the current input has the context of previous inputs but not the future. Bidirectional RNN, on the other hand, duplicates the RNN processing chain so that the inputs are processed in both forward and backward orders to enable looking into future contexts as well.

The input trajectories (to the meta-learner) are selected from the RL trajectory database (that are generated by the RL agent interacting with the current RL environment) dynamically at runtime. We chose the top M trajectories that have resulted in the highest rewards so far because the experimental results show that the trajectories with lower rewards are unhelpful or even harmful. Intuitively, those lower-reward trajectories are generated with a random policy or a poorly trained policy, so they are not representative of the workloads.

The output from the bidirectional RNN of the meta-learner is an embedding that is used to fingerprint/represent the <application, environment> pair with which the base-learner is interacting. As shown in Fig. 5, the generated embedding based on past experience (i.e., the episodes previously explored by the base-learner) is fed to the base-learner as part of the input at each time step. Since we adopted as our base-learner the RL design from FIRM [47], which is an actor-critic RL algorithm, the embedding is taken by the actor network.

Interpreting Embeddings from Systems Perspective. The environment-specific embedding is able to differentiate one <application, environment> pair from another and thus guides the base-learner to adapt to the new environment. Fig. 6 visualizes the key idea of embedding. The spatial and temporal characteristics of the workloads are encoded and mapped onto a low-dimensional latent vector space by the embedding layer. Workloads with similar characteristics are projected to locations that are close to each other on that vector space. By calculating the cosine similarity between any two generated vectors (i.e., embeddings), we can get a monotonic similarity measure. To help understand how generated embeddings can represent spatial and temporal characteristics, we selected RL trajectories from <application, environment> pairs with human-detectable different performance sensitivities or load patterns, and then the plotted embedding projection shows that indeed similar workloads are closer to each other when comparing cosine similarities of their embeddings. In Fig. 6 (upper), the sensitivity of application performance to different resource allocations is shown in the heatmaps to illustrate the spatial characteristics, with the X-axis being CPU cores and the Y-axis being allocated RAM. Darker colors represent worse performance in terms of application request-serving latency. In Fig. 6 (lower), the application load-per-second time series are plotted to represent the temporal characteristics. Again, workloads with similar patterns are projected to adjacent locations in the output vector space.

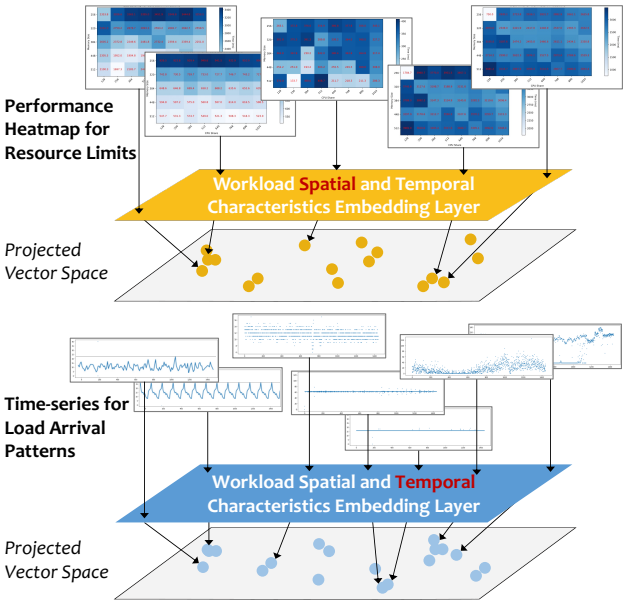


Figure 6: An example illustrating the idea of workload embedding for encoding spatial and temporal characteristics from a systems perspective. *The yellow points (upper figure) indicate that workloads with similar performance sensitivities (to resource allocations) are projected to locations near each other in the embedding vector space. The blue points (lower figure) show that workloads with similar load arrival patterns are projected to adjacent locations in the embedding vector space. The similarity metric used is cosine similarity.*

Meta-learner Training. During the training of the meta-learner, both the meta-learner and base-learner model parameters are updated. After each RL episode, the loss value is generated by the base-learner and is backward-propagated to update the model parameters in the base-learner. Since the meta-learner is trained across a distribution of environments, the total loss of all sampled environments in the training dataset is used to update the model parameters in the meta-learner. In the end, the trained meta-learner is capable of abstracting the individuality of each <application, environment> pair; the trained base-learner is a shared RL model that is able to generate optimal workload autoscaling policies conditioned on the workload embeddings provided by the meta-learner. The base-learner can be used as a starting point and as the basis for fine-tuning a specific novel <application, environment> pair in the inference stage.

Meta-learner Inference. After the meta-learner is trained, the meta-learning model is able to adapt the base-learner to a new <application, environment> pair that has never been encountered during training. Note that even though the new environment has never been encountered during training, it comes from the same distribution as, or shares similar patterns with, the encountered ones, so that transferring is still possible [12, 20, 39, 43]. The adaptation process only requires limited exposure to the new environment. Therefore, AWARE simply samples RL episodes and runs the meta-learner to generate the workload embedding. With the workload embedding, the base-learner can be continuously trained to learn the workload autoscaling policy for the new <application, environment> pair. The meta-learner model parameters are fixed during the inference stage.

Algorithm 1 RL agent lifecycle transition management for bootstrapping and triggering of online retraining. Four status codes `INITIALIZED`, `ONLINE`, `OFFLINE`, and `SERVING` stand for agent-initialized, online training, offline training, and online policy-serving, respectively.

Require: Rewards $R = [r_t]_{t \in T}$, User Profile P

1: **procedure** OBSERVEANDTRIGGER(R, P)
2: $stage \leftarrow$ INITIALIZED
3: **while** $True$ **do**
4: **if** $state$.equal(INITIALIZED) **then**
5: **if** P.BOOTSTRAP == $True$ **then**
6: $stage \leftarrow$ OFFLINE ▷ Bootstrapping
7: **else**
8: $stage \leftarrow$ ONLINE ▷ Skip Bootstrapping
9: **end if**
10: **else if** $state$.equal(OFFLINE) **then**
11: **if** avg(R) $\geq P.T_{online}$ **then**
12: $stage \leftarrow$ ONLINE
13: **end if**
14: **else if** $state$.equal(ONLINE) **then**
15: **if** avg(R) $\geq P.T_{serving}$ & std(R) $\leq P.T_{var}$ **then**
16: $stage \leftarrow$ SERVING
17: **end if**
18: **else if** $state$.equal(SERVING) **then**
19: **if** avg(R) $< P.T_{serving}$ $\|$ std(R) $> P.T_{var}$ **then**
20: $stage \leftarrow$ ONLINE
21: **end if**
22: **end if**
23: **end while**
24: **end procedure**

3.3 Incremental Retraining

When deploying the RL agent in a production system, one needs to ensure that the policy behaves as expected and scales to the workload in production. AWARE leverages continuous monitoring to detect any anomalous behavior and trigger retraining when needed. Alg. 1 describes how AWARE's RL retraining module (❸ in Fig. 4) manages the lifecycle of the agent and enables incremental retraining at runtime (lines 14–21). The input to the retraining module includes (a) the user profile specifying the configuration, and (b) recent rewards pulled from the RL trajectory database. When the mean and the standard deviation of the recent rewards satisfy the threshold-based condition (i.e., agent performance is bounded to a target value), the agent enters the policy-serving stage; otherwise, the agent enters the policy-training stage. Retraining of the RL agent is by online interaction with the RL environment. As discussed in §3.4, non-RL-based approaches

(i.e., HPA and VPA) can be used as a fallback option for RL agents when high-stakes applications want to keep the RL agent in the offline mode during retraining.

3.4 Bootstrapping

The policy at the early RL training stages could be worse than the baseline approaches. For example, overprovisioning leads to low resource utilization, while under-provisioning results in SLO violations. For production workloads, especially high-stakes applications, such suboptimal actions are not acceptable. In AWARE, an RL bootstrapper (❹ in Fig. 4) has been designed to combine offline and online RL training. If the user specifies enabling bootstrapping (as shown in lines 4–9 Alg. 1), the offline mode will be turned on first. AWARE will then use Kubernetes HPA [25] (which is a threshold-based approach) for horizontal workload autoscaling, and use Kubernetes VPA [15] (which adjusts resource limits based on history profile) for vertical workload autoscaling. Note that HPA and VPA can also be used as a fallback option for RL when high-stakes applications want to keep the RL agent in the offline mode during retraining, as discussed in §3.3.

In the offline mode, the RL bootstrapper intercepts the request-response path between the agent and the RL API gateway and replaces the RL agent with the fallback controller to react to the received states and generate actions at each time step. The RL API gateway then takes the received action for execution, and the resulting behavior is the same as when workloads are managed by HPA and VPA. The RL agent samples trajectories from the trajectory database for offline policy training. To overcome extrapolation errors whereby previously unseen state-action pairs are erroneously estimated, we apply a state-conditioned generative model to combine with the critic network for producing previously seen actions [14]. In the online training mode, the agent will then directly interact with the RL environment through the API gateway.

4 Implementation

4.1 Kubernetes MPA

We propose our own design and implementation of multi-dimensional Pod autoscaling because the current HPA and VPA controllers are independent of each other and can lead to a large number of tiny Pods [16]. Google MPA [10] is a pre-GA beta version product that offers an integrated solution for HPA and VPA, but it is not open-sourced and does not support custom recommenders. In AWARE, the MPA framework combines the actions of vertical and horizontal autoscaling but separates the actuation from the controlling algorithms. As shown in Fig. 7, there are three controllers (i.e., a recommender, an updater, and an admission controller) and an MPA API (i.e., a CRD object [24]) that connects the autoscaling recommendations to actuation.

The multidimensional scaling algorithm is implemented in the recommender mostly by importing HPA and VPA libraries to serve as the fallback option for RL-based approaches. The

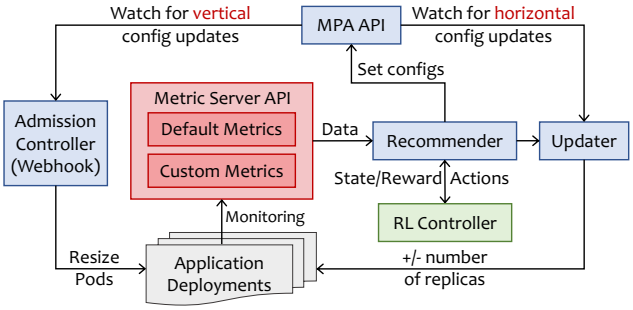


Figure 7: MPA design overview and integration with RL.

metrics required by the algorithm are collected from the Kubernetes Metrics Server, including default metrics such as container resource utilizations and custom metrics such as application throughput or latency. The scaling decisions derived from the recommender are stored in the MPA object as scaling configurations. The updater and the admission controller retrieve those updated configurations from the MPA object and then actuate them as vertical and horizontal actions on the application Deployments. The separation of action actuation from scaling decision generation allows developers to replace the default recommender with the alternative recommender, i.e., the RL controller. The implementation is in Go and at the stage of releasing to the Kubernetes upstream as well.

4.2 Integration with RL

The creation of MPA is through declarative YAML files. To integrate MPA with RL agents, one needs to specify a custom recommender to replace the default recommender (HPA+VPA). After an MPA is initialized for the application deployment, an MPA wrapper is created as a shim layer to communicate with the RL agent through RPCs. We follow the "agent-centric" pattern of request-response interaction advocated by OpenAI Gym [44]. The exposed interfaces include (a) `init()` (for initializing the RL environment), (b) `state = reset()` (for resetting the environment at the beginning of each RL episode), and (c) `state, reward = step(action)` (for RL agent stepping). When the MPA wrapper receives an action through the RPC request, it first translates the action to vertical and horizontal scaling configurations and writes to the MPA object. We deploy Prometheus [7], the standard monitoring service in Kubernetes, to export default and custom metrics from the application Deployment. The wrapper then queries the Prometheus service for real-time metrics and translates to RL states and rewards. Finally, the wrapper sends the metrics back to the agent through the RPC response. The MPA wrapper is implemented in Python.

4.3 Meta-learning-based RL-serving

AWARE's meta-learning-based RL agent management framework is implemented in Python. Both the base-learner (adopted from FIRM [47]) and the meta-learner are implemented using PyTorch [13]. The meta-learner is essentially a bidirectional two-layer RNN followed by two fully connected

layers with the ReLU activation function. We chose the trajectory bundle size to be 20 for the fastest adaptation with the fewest trajectories according to the sensitivity analysis. Each RNN hidden layer consists of 256 neurons, and the fully connected layers consist of 256 and 64 neurons. We chose two layers and an embedding size of 64 because adding more layers and hidden units does not increase performance in our experiments; instead, it slows down training speed significantly. We used the Adam optimizer for parameter updates.

RL trajectories are saved to InfluxDB [21], an open-source time-series database that is built to handle metrics with time-stamped data. Recent rewards, sampled RL trajectories for offline base-learner training, and the inputs for embedding generation are all pulled from the trajectory database by using the InfluxDB Python client library.

AWARE provides a simple and declarative user interface for RL pipeline developers, which is consistent with Kubernetes' way of creating and managing objects in the cluster. To specify the targets for the workloads, i.e., resource utilization targets and the application SLO (if there is one), users only need to provide a YAML file following the definition template. Both application latency and throughput SLOs are currently supported. In addition, users can also specify the thresholds for RL rewards and whether or not to enable bootstrapping in the YAML file, which constructs the profile used in Alg. 1.

5 Evaluation

Our experiments addressed the following research questions:

§5.2 Does AWARE provide fast model adaptation to new workloads? What is the value of meta-learning?

§5.3 How does AWARE perform in online policy-serving when workload updates or load changes occur?

§5.4 How does AWARE perform in the early stages of policy training, compared to RL agents without bootstrapping?

5.1 Experimental Setup

We implemented an application generator capable of generating a large number of synthetic applications by combining the 16 selected representative production application segments [11] (discussed in §2.3 as well) based on random sampling with replacement from the segment pool. Each segment represents the smallest granularity of common workloads in cloud datacenters. In addition, each segment has to be associated with its own inputs to simplify load generation (e.g., the image manipulation workloads come with random images). The generator also comes with setup and tear-down scripts for all external services each segment uses (e.g., databases or messaging queues). Overall, we generated 1000 unique applications, deployed them as Deployments in a Kubernetes cluster of 11 two-socket physical nodes, and ran an RL-based multidimensional autoscaler with each application. Each server consists of 56–192 CPU cores and RAM that vary from 500 GB to 1000 GB. Seven of the servers use Intel x86 Xeon E5 processors, and the remaining ones use IBM ppc64 Power8 and Power9 processors.

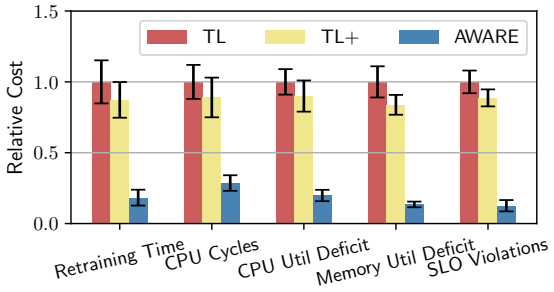


Figure 8: RL agent retraining cost and performance comparison of AWARE, transfer learning (TL), and transfer learning with augmented features (TL+).

While it would be impossible to cover all cloud workloads, the selected production workload segments should enable the generation of a large number of synthetic cloud workloads with varying resource consumption profiles. In the future, the number of implemented segments can easily be extended if specific workload profiles are missing. We refer to the open-source artifact for additional details on the generator implementation. With the same datacenter workload traces [65] discussed in §2.3 with respect to RL agent training and policy-serving, we divided the 1000 generated application pool with the 8:2 ratio. The 800 applications with varied workloads are used to train the meta-learner, while the remaining 200 applications are used to evaluate the adaptability. The total runtime is ∼60 days, and the meta-learner training time is ∼312 hours on an Intel(R) Xeon(R) E5-2695 processor.

The RL formulation and design (in the base-learner) are adopted from FIRM [47] (as mentioned in §2.3). As an end-to-end evaluation, Fig. 1 shows that, compared to AWARE, the RL-based autoscaler FIRM by itself suffered from poor performance during the initial training stage (i.e., Stage ①, which demonstrates the benefit of AWARE's bootstrapping mechanism), online policy-serving performance degradation (i.e., Stage ②, which demonstrates the benefit of the online retraining triggering mechanism), and slow adaptation with non-trivial retraining (i.e., Stage ③, which demonstrates the benefit of meta-learning). We then present the evaluation results related to each research question in §5.2–§5.4.

5.2 Fast Adaptation

To study adaptability to new workloads, we compared AWARE with the existing transfer learning approach. FIRM [47] leverages transfer learning to train an RL agent for a new service based on previous RL experience gained when training the RL agent for a known service. In the transfer-learning-based approach (TL), the model parameters (weights) are shared between the agents managing the known workload and the new workload. We also compared AWARE with a novel approach (TL+) based on transfer learning that includes additional spatial and temporal features in the RL states, since the meta-learner in AWARE is trained to output an embedding to represent the spatial and temporal characteristics of the <application, environment> pair. We

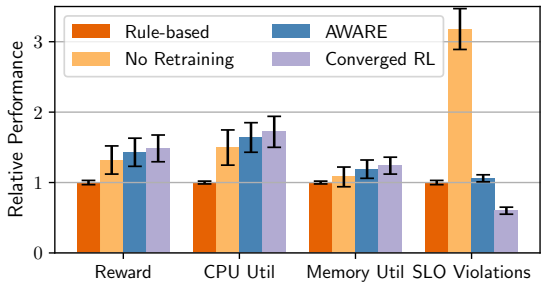


Figure 9: RL agent online policy-serving performance comparison of AWARE, no retraining, the rule-based method, and the agent with the converged RL policy. *In the comparison of reward and CPU/memory utilization, the higher, the better, while a lower number of SLO violations is better.*

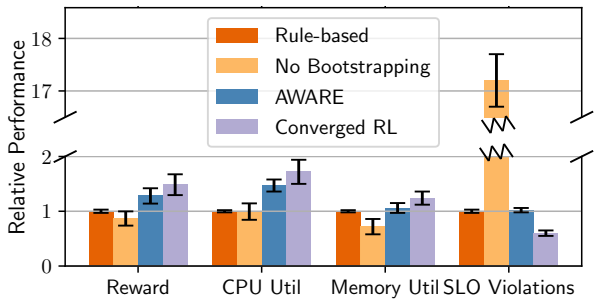


Figure 10: RL agent training performance comparison of AWARE, no bootstrapping, the rule-based method, and the agent with the converged RL policy. *In the comparison of reward and CPU/memory utilization, the higher, the better, while a lower number of SLO violations is better.*

used the widely used ARIMA model [19] to generate the predicted load for the next time step (i.e., temporal feature) and recorded a table mapping from resource allocation to performance (i.e., spatial feature). We performed A/B tests in which the workload traces were the same, but the recommender in MPA was replaced with TL, TL+, and AWARE, which drove the horizontal and vertical scaling of the workload. We repeated the A/B test 100 times. In each test, we randomly selected a workload from the pool and trained the RL agent to convergence. We then randomly selected 10 other different workloads from the pool for adaptivity evaluation. We measured the retraining time, CPU cycles involved in retraining, utilization deficit (compared to the converged RL policy), and SLO violations.

Fig. 8 shows that AWARE adapted 5.5× and 4.6× faster (saved 68–72% CPU cycles) than TL and TL+, respectively. During the adaptation period, TL+ had 4.6× and 6.2× higher CPU and memory utilization deficit compared to AWARE while AWARE reduced SLO violations by 7.1×. TL+ encodes additional spatial and temporal features, but each state is still a stateless snapshot of the running workload. Additional features (i.e., the table and the ARIMA output) greatly increase the state space. Meta-learning, on the other hand, offers a systematic and automated way of learning how to differentiate the workloads well and outputs a low-dimensional embedding to be used by the base-learner.

5.3 Online Policy-serving

To evaluate the online policy-serving performance when facing workload updates and load changes (described in §2.3), we compared AWARE with (a) a rule-based approach, (b) an RL agent without continuous monitoring and retraining, and (c) an RL agent with the converged policy. For the rule-based approach with manual scaling, we measured the maximum CPU utilization when the SLO was met, and set it as the threshold for HPA. We used the default Auto mode [15] for VPA. We performed the same style of A/B tests 100 times and replaced the MPA recommender with the four approaches. In each A/B test, we randomly selected a workload from the pool, trained the RL agent to convergence, and injected a series of the seven random instability scenarios introduced in §2.3. We then measured the average reward, CPU/memory utilization, and the number of SLO violations during the time until the agent managed by AWARE converged. Fig. 9 shows that AWARE had 9.6% and 14.8% higher CPU and memory utilization, and reduced SLO violations by 3.1× compared to the RL agent without retraining (the second-best approach), resulting in 8.6% higher per-episode reward. Compared to the converged RL policy, AWARE had a 3.6% lower average per-episode reward because we set the retraining threshold to be 5, corresponding to a 2.6% reward degradation. Sensitivity analysis showed that AWARE converged to the no-retraining baseline as the threshold increased while a smaller than 5 threshold led to constant retraining with no policy serving.

5.4 Bootstrapping

To study how much bootstrapping helps reduce the cost of early-stage RL training, we compared AWARE with (a) the rule-based approach (same as in Fig. 9), (b) an RL agent without bootstrapping, and (c) an RL agent with the converged policy. Since the per-episode reward achieved by the rule-based autoscaler is around 130 (translated from the measured utilization and performance), we set the bootstrapping threshold in AWARE to 130. In the sensitivity analysis, we observed that a higher threshold led to endless bootstrapping driven by the rule-based autoscaler (since the measured reward is always lower than the threshold), while the lower the threshold, the more performance degradation AWARE had during its early-stage training. A threshold of 0 basically converges to the learning curve without AWARE bootstrapping (i.e., no offline learning). We performed A/B tests 100 times and replaced the MPA recommender with the four approaches. In each A/B test, we randomly selected a workload from the pool and trained the RL agent to convergence (with or without bootstrapping). We then measured the average reward, CPU utilization, memory utilization, and the number of SLO violations during the time until the RL agent converged. Fig. 10 shows that AWARE had 47.5% and 39.2% higher CPU and memory utilization, respectively, and reduced SLO viola-

tions by a factor of 16.9× compared to the RL agent without bootstrapping (the second-best approach), resulting in 47.3% higher average per-episode reward before convergence.

6 Related Work

RL Training and Model-serving Frameworks. Ray [41] is an open-source distributed execution framework that facilitates RL model training and serving by making it easy to scale an RL application and schedule distributed runs to efficiently use all resources (i.e., CPU, memory, or GPU) available in a cluster. Amazon SageMaker [1] uses the Ray RLlib library that builds on the Ray framework to train RL policies. SageMaker also provides cloud services that help build and deploy ML models (e.g., data processing and model evaluation). RLzoo [9] is an RL library that aims to make the development of RL agents efficient by providing high-level yet flexible APIs for prototyping RL agents. RLzoo also allows users to import a wide range of RL agents and easily compare their performance. Park [33] provides 12 representative RL environments in the field of systems and networking (e.g., job scheduling) for developing and evaluating RL algorithms. Genet [60] is an RL training framework for learning better network adaptation policies. Genet leverages curriculum learning [42], which aims to sequence tasks to achieve the best performance on a specific final task instead of quickly adapting to a new task within a small number of gradient descent steps.

RL in Production. Panzer *et al.* [45] provide a survey of existing RL applications in production system domains, including resource scheduling. They summarize the implementation challenges and generalizability of simulation-trained RL models. SOL [58] is an extensible framework for developing ML/RL-based controllers for tasks such as core frequency scaling. SOL is complementary to AWARE, which can further guarantee that the RL agent operates safely under various realistic issues, including bad data and external interference like resource unavailability. SIMPPO [36, 48] provides a scalable framework based on the mean-field theory that enables multiple RL agents to coexist in a shared multi-tenant environment. Autopilot [50] is a workload autoscaler used at Google that leverages multi-armed bandits (i.e., the simplest version of RL) to choose a variant of the sliding window algorithms that historically would have resulted in the best performance for each job. In its essence, it is still a heuristic mechanism and has been shown [59] to suffer from poor system stability because of inaccurate estimation of horizontal concurrency; it can also result in a large number of tiny Pods [16] due to the independence between horizontal and vertical scaling.

7 Discussion and Future Challenges

Extension to Other System Domains. AWARE is a general and extensible framework that can be applied to other systems management tasks (e.g., congestion control or job scheduling). To apply it to a new domain, one needs to (a) replace the RL environment by implementing the provided environment wrapper interface; and (b) provide a default non-RL-based agent for the RL bootstrapper. We leave the study of the performance in other system domains to the future.

Out-of-distribution Workloads. AWARE provides the opportunity to quickly customize the model to specific workloads. However, out-of-distribution <application, environment> pairs still require training because meta-learning assumes that all pairs, including the unseen cases, are inherently within the learned distribution [20] (e.g., in terms of service request arrival patterns or sensitivity to resource allocation). Given the diversity of workloads in the cloud datacenter (used in the training dataset), the meta-learner and the shared base-learner can be continuously trained, and out-of-distribution cases are covered eventually. Meanwhile, with offline RL training, users can still benefit from the heuristics-based solution used as the fallback option. One limitation of our experiment was that the generated applications might not have covered all possible cloud workloads. However, application segments can easily be extended in the synthetic application generator if specific workload profiles are missing (§5.1).

On-policy RL Algorithms. When RL agents are being bootstrapped at the initial stage, off-policy RL agents (such as DDPG [29, 47] and DQN [59, 62]) can be trained directly using the collected RL trajectories. However, on-policy RL agents (such as PPO [48]) require trajectories generated from their own policy. One potential way to train on-policy RL agents offline would be to build a simulator based on the collected trajectories, which would essentially map resource allocation to workload performance and system metrics. A balanced experience replay scheme [27] could potentially be applied for locating near-on-policy samples from the simulator constructed based on the offline dataset. Instead of drawing trajectories from the trajectory database (as in §3.4), the RL base-learner can interact with the simulator for bootstrapping.

8 Conclusion

This paper explored the challenges of applying RL in workload autoscaling in production cloud platforms. We presented a general and extensible framework for deploying and managing RL agents in production systems. To demonstrate the framework, we implemented AWARE for automating RL-based workload autoscaling in Kubernetes and experimentally showed (a) the benefits of leveraging meta-learning for fast model adaptation, and (b) how the design of AWARE ensures the stable and robust online performance of RL models.

Acknowledgments

We thank the anonymous reviewers and our shepherd Xiaosong Ma for their valuable comments that improved the paper. This work is partially supported by the National Science Foundation (NSF) under grant No. CCF 20-29049; and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDAI). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM.

Availability

We provide an open-source implementation of AWARE at https://gitlab.engr.illinois.edu/DEPEND/aware.

References

[1] AWS. Amazon SageMaker. https://aws.amazon.com/sagemaker/, 2022. Accessed: 2022-11-23.

[2] AWS. AWS autoscaling documentation. https://docs.aws.amazon.com/autoscaling/index.html, 2022. Accessed: 2022-11-23.

[3] Azure. Azure autoscale. https://azure.microsoft.com/en-us/features/autoscale/, 2022. Accessed: 2022-11-23.

[4] Subho Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Inductive-bias-driven reinforcement learning for efficient scheduling in heterogeneous clusters. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, pages 629–641, Cambridge, MA, USA, 2020. PMLR.

[5] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Machine learning for load balancing in the Linux kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (ApSys 2020)*, pages 67–74, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Google Cloud. Google cloud load balancing and autoscaling. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling, 2022. Accessed: 2022-11-23.

[7] CNCF. Prometheus. https://prometheus.io/, 2022. Accessed: 2022-11-23.

[8] Sundar Dev, David Lo, Liqun Cheng, and Parthasarathy Ranganathan. Autonomous warehouse-scale computers. In *ACM/IEEE 57th Design Automation Conference (DAC 2020)*, pages 1–6, 2020.

[9] Zihan Ding, Tianyang Yu, Hongming Zhang, Yanhua Huang, Guo Li, Quancheng Guo, Luo Mai, and Hao Dong. Efficient reinforcement learning development with RLzoo. In *Proceedings of the 29th ACM International Conference on Multimedia (MM 2021)*, pages 3759–3762, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Google GKE Documentation. Configuring multidimensional Pod autoscaling in GKE. https://cloud.google.com/kubernetes-engine/docs/how-to/multidimensional-pod-autoscaling, 2022. Accessed: 2022-11-23.

[11] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2021.

[12] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, pages 1126–1135. JMLR.org, 2017.

[13] The Pytorch Foundation. PyTorch. https://pytorch.org/, 2022. Accessed: 2022-11-23.

[14] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, pages 2052–2062. PMLR, 2019.

[15] GitHub. Vertical Pod Autoscaling (VPA) in Kubernetes. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler, 2022. Accessed: 2022-11-23.

[16] Google GKE. Challenges of scaling kubernetes Pods horizontally and vertically. https://cloud.google.com/blog/topics/developers-practitioners/scaling-workloads-across-multiple-dimensions-gke, 2022. Accessed: 2022-11-23.

[17] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pages 620–629, 2018.

[19] Siu Lau Ho and Min Xie. The use of ARIMA models for reliability forecasting and analysis. *Computers & Industrial Engineering*, 35(1-2):213–216, 1998.

[20] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(09):5149–5169, 2022.

[21] InfluxData. InfluxDB. https://github.com/influxdata/influxdb, 2022. Accessed: 2022-11-23.

[22] Nathan Jay, Noga H. Rotman, P. Godfrey, Michael Schapira, and Aviv Tamar. Internet congestion control via deep reinforcement learning. *In Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, 32, 2018.

[23] Sara Kardani-Moghaddam, Rajkumar Buyya, and Kotagiri Ramamohanarao. ADRL: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds. *IEEE Transactions on Parallel and Distributed Systems (TPDS 2020)*, 32(3):514–526, 2020.

[24] Kubernetes. Extending Kubernetes API with custom resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources, 2022. Accessed: 2022-11-23.

[25] Kubernetes. Horizontal Pod Autoscaling (HPA) in Kubernetes. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2022. Accessed: 2022-11-23.

[26] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[27] Seunghyun Lee, Younggyo Seo, Kimin Lee, Pieter Abbeel, and Jinwoo Shin. Offline-to-online reinforcement learning via balanced replay and pessimistic Q-ensemble. In *Proceedings of the 5th Conference on Robot Learning (CoRL 2021)*, pages 1702–1712. PMLR, 2021.

[28] Xu Li, Feilong Tang, Jiacheng Liu, Laurence T. Yang, Luoyi Fu, and Long Chen. AUTO: Adaptive congestion control based on multi-objective reinforcement learning for the satellite-ground integrated network. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC 2021)*, pages 611–624. USENIX Association, 2021.

[29] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*, 2016. https://arxiv.org/abs/1509.02971.

[30] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*, pages 450–462, 2015.

[31] Yiqing Ma, Han Tian, Xudong Liao, Junxue Zhang, Weiyan Wang, Kai Chen, and Xin Jin. Multi-objective congestion control. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys 2022)*, EuroSys '22, pages 218–235, New York, NY, USA, 2022. Association for Computing Machinery.

[32] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNet 2016)*, pages 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems (NeurIPS 2019)*, 32, 2019. https://proceedings.neurips.cc/paper/2019.

[34] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2017)*, pages 197–210, New York, NY, USA, 2017. Association for Computing Machinery.

[35] 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 2019)*, pages 270–288, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Weichao Mao, Haoran Qiu, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, and Tamer Başar. A mean-field game approach to cloud resource management with function approximation. In *Proceedings of the 36th Conference on Advances in Neural Information Processing Systems (NeurIPS 2022)*, volume 36, pages 1–12, New Orleans, LA, USA, 2022. Curran Associates, Inc.

[37] Jason Mars and Lingjia Tang. Whare-Map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 2013)*, pages 619–630, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural*

Information Processing Systems (NeurIPS 2013), pages 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[39] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2018. https://openreview.net/forum?id=B1DmUzWAW.

[40] Jun Morimoto and Kenji Doya. Robust reinforcement learning. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems (NeurIPS 2000)*, volume 13. MIT Press, 2000. https://proceedings.neurips.cc/paper/2000.

[41] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI 2018)*, pages 561–577, USA, 2018. USENIX Association.

[42] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(1), 2022. https://jmlr.org/papers/volume21/20-212/20-212.pdf.

[43] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[44] OpenAI. OpenAI Gym documentation. https://www.gymlibrary.dev/, 2022. Accessed: 2022-11-23.

[45] Marcel Panzer and Benedict Bender. Deep reinforcement learning in production systems: A systematic literature review. *International Journal of Production Research*, 60(13):4316–4341, 2022.

[46] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta. Robust adversarial reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, pages 2817–2826. JMLR.org, 2017.

[47] 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 2020)*, pages 805–825, Berkeley, CA, USA, November 2020. USENIX Association.

[48] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. SIMPPO: A scalable and incremental online learning framework for serverless resource management. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pages 306–322, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *Proceedings of the 12th International Conference on Cloud Computing (CLOUD 2019)*, pages 329–338, 2019.

[50] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at Google. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys 2020)*, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3342195.3387524.

[51] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML 2015)*, pages 1842–1850. JMLR.org, 2016.

[52] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[53] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*, pages 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[54] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *The 18th Workshop on Hot Topics in Operating Systems (HotOS 2021)*, pages 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[55] Ilya Sutskever, James Martens, and Geoffrey E. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML 2011)*, 2011. https://icml.cc/2011/papers/524_icmlpaper.pdf.

[56] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. Reinforcement learning for datacenter congestion control. *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2022)*, 36(11):12615–12621, Jun. 2022.

[57] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*, pages 384–394, USA, 2010. Association for Computational Linguistics.

[58] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. SOL: Safe on-node learning in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*, pages 622–634, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pages 16–30, New York, NY, USA, 2022. Association for Computing Machinery.

[60] Zhengxu Xia, Yajie Zhou, Francis Y. Yan, and Junchen Jiang. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 397–413, New York, NY, USA, 2022. Association for Computing Machinery.

[61] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: A randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020)*, pages 495–511, 2020.

[62] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *IEEE 39th International Conference on Distributed Computing Systems (ICDCS 2019)*, pages 122–132, Washington, DC, USA, 2019. IEEE Computer Society.

[63] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. FaaSRank: Learning to schedule functions in serverless platforms. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2021)*, pages 31–40, Washington, DC, USA, 2021. IEEE Computer Society.

[64] Kuo Zhang, Peijian Wang, Ning Gu, and Thu D. Nguyen. GreenDRL: Managing green datacenters using deep reinforcement learning. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022)*, pages 445–460, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021)*, pages 724–739, New York, NY, USA, 2021. Association for Computing Machinery.

Nodens: Enabling Resource Efficient and Fast QoS Recovery of Dynamic Microservice Applications in Datacenters

Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, Minyi Guo
Department of Computer Science and Engineering, Shanghai Jiao Tong University

Abstract

Current microservice applications always meet with load and call graph dynamics. These dynamics can easily lead to inappropriate resource allocation for microservices, and further lead to Quality-of-Service (QoS) violations of applications. However, current microservice management works are incapable to handle these dynamics, mainly due to the execution blocking effect among microservices. We therefore propose Nodens, a runtime system that enables fast QoS recovery of the dynamic microservice application, while maintaining the efficiency of the resource usage. Nodens comprises a *traffic-based load monitor*, a *blocking-aware load updater*, and a *resource-efficient query drainer*. The load monitor periodically checks microservices' network bandwidth usage and predicts the monitored loads based on it. The load updater updates the actual "to-be-processed" load of each microservice to enable fast resource adjustment. The query drainer allocates just-enough excessive resources for microservices to drain the queued queries, which can ensure the QoS recovery time target. Our experiments show that Nodens can reduce the QoS recovery time by 12.1X with only the excessive resource usage of 6.1% on average, compared to the state-of-the-art microservice management systems.

1 Introduction

User-facing applications are evolving towards the microservice architecture, with which the microservices communicate through the network [25, 41] and are able to scale independently [1, 5, 9]. The dependencies of the microservices can often be denoted by a Directed Acyclic Graph (DAG) [31, 32], each node represents a microservice and each edge represents the call dependency [24, 38]. Moreover, a production microservice application often has multiple call graphs [31, 32], as users have different query patterns. Figure 1 shows an example dependency graph and two call graphs that handle different user queries.

In these applications, the load of each microservice change dynamically, because 1) the load of the entire application may change over time due to the diurnal, irregular, and bursty load patterns (referred to be "load dynamic") [3, 19, 28, 40], and 2) the percentages of queries that go to different call graphs may change over time (referred to be "call graph dynamic") [34]. We analyze the open-sourced production traces [2, 8], and find that the load variation is 30% on average (up to 7.5X), and the percentages of queries that go to different call graphs also vary by 15% on average (up to 70%).

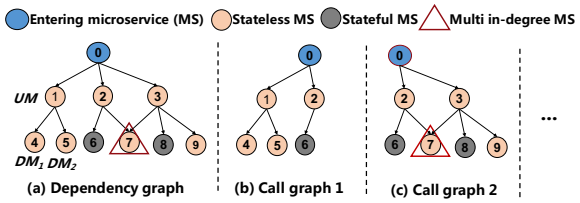


Figure 1: An example microservice dependency graph and two call graphs.

While it is crucial to ensure the required Quality-of-Service (QoS) of user-facing applications [20, 30, 39], prior works [22, 38, 41, 44] fail to handle these dynamic applications. Most prior works periodically check the load of each microservice, and adjust the resource allocation of each microservice based on the monitored load. They are incapable for the current dynamic microservice applications for two main reasons.

As for the first reason, the monitored load of a microservice may not be its real "to-be-processed" load due to the cascade call relationship. Many queries may be blocked by its upstream microservices. A microservice may not be allocated enough computation resources in this case. Worse, there is a lag in noticing the load increase. For instance, if the monitoring period is 1 second, the lag can be 1 second as well. However, the short lag may result in the long QoS violation, as a great many of queries may queue up at the microservice. A very long time is needed to adjust resources and drain up the query queue, when the microservice's resource is allocated based on the monitored load. Our experiments show that the QoS can be recovered in as long as 84.4 seconds.

Some other prior work [18, 23, 25, 45] predict the QoS and

adjust the resource allocation beforehand. They assume all the queries go through all the microservices, thus are not able to handle the dynamic loads due to the variation of call graphs.

An intuitive solution is calculating the actual "to-be-processed" queries of each microservice based on the dependency graph and the call graphs, and adjusting their resources accordingly. However, it does not work because we find that the queries of a microservice may also be blocked by other microservices besides of its upstream microservices in the dependency graph. For instance, microservice-2 in Figure 1 may be blocked by microservice-4 or microservice-5 that do not call it in the dependency graph. This happens when microservice-0 calls microservice-1 and microservice-2 in a fixed order and the resource allocation of microservice-4 or microservice-5 is insufficient.

An appropriate solution should be able to capture all the potential "blocking" relationships, and be able to drain up the query queues due to the monitoring lag. We therefore define an *execution blocking graph* that captures all the superior microservices that may block a microservice, based on which we further propose a runtime system named **Nodens**[1] that enables fast QoS recovery, if the loads of some microservices suddenly increase due to the two types of dynamics. Note that the execution blocking graph is not the same as the microservice dependency graph.

Nodens comprises a *traffic-based load monitor*, a *blocking-aware load updater*, and a *resource-efficient query drainer*. For each microservice, the load monitor periodically checks the input network traffics, and predicts the current monitored load of the microservice based on the traffics. This method is much faster than obtaining the load information from the microservices' interfaces, enabling earlier resource allocation adjustment. The load updater updates the execution blocking graph with the monitored load obtained from the monitor, and estimates the actual "to-be-processed" load of each microservice. The query drainer adjusts the CPU resource allocated to each microservice based on the actual loads of the microservices and the queued queries during the previous process, in order to quickly recover the QoS.

This paper makes three main contributions.

- **Comprehensive analysis of current methods to handle microservice dynamics.** The insights obtained from the analysis identify the opportunities to enable fast QoS recovery when dealing with microservice dynamics.
- **The design of a method to update actual loads of microservices under execution blocking effect.** We construct the execution blocking graph based on microservice dependencies, with which we can update actual loads of microservices under the blocking effect.
- **The design of a policy to drain the queued queries during the resource adjustment process.** The policy allocates excessive resources for microservices to satisfy the QoS recovery time target, while maintaining the resource efficiency.

We evaluate Nodens with our benchmarks on an eight-node cluster. The experimental results show that Nodens can reduce the QoS recovery time by 12.1X with only the over-provisioned resources of 6.1% on average, compared to the state-of-the-art microservice management systems.

[1]The source code is available at https://github.com/shijiuchen/Nodens.

2 Related Work

There has been some related work on ensuring the QoS of user-facing applications.

2.1 Reactive Microservice Management

Reactive microservice management systems periodically monitor the state (e.g., load or latency) of each microservice, and adjust the resource allocation of each microservice based on the state.

Heuristic methods: SHOWAR [17], PEMA [27], Astraea [44], and ATOM [26] designed heuristic approaches to conduct horizontal or vertical scaling for CPU or GPU microservices based on the resource utilization and response latency. These heuristic methods can determine the resource allocation for microservices in a quick way, but hard to achieve the near-optimal values.

Machine Learning (ML) based methods: Nautilus [22] used Reinforcement learning (RL) as feedback to tune microservices' resources based on the application's response latency. FIRM [38] identified the critical microservices which caused QoS violations, and used RL as feedback to adjust resources for each microservice based on tail latencies, to guarantee the QoS of the application. ELIS [41] utilized the bayesian optimization algorithm with tail latencies as input to recycle the over-provisioned resources and then allocate just-enough resources to critical microservices. These ML-based methods can allocate near-optimal resources for microservices, but are slower due to incremental training under microservice dynamics.

Moreover, above reactive methods all have long QoS recovery time when handling dynamics of microservices, which is caused by the long monitoring interval, and the load blocking under the cascade call relationship among microservices.

2.2 Proactive Microservice Management

Proactive microservice management systems predicted the performance and resource allocation for microservices based on historical data. Seer [25] and Sage [23] used ML-based methods to predict the microservices that cause QoS violations based on the latency metrics, and increase the allocated resources for them. Sinan [45] and DeepRest [18] utilized

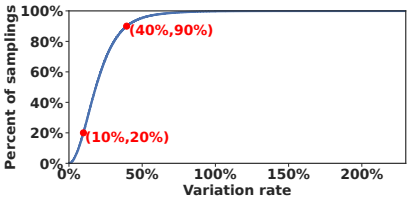


Figure 2: Load variation of top 20 microservice applications.

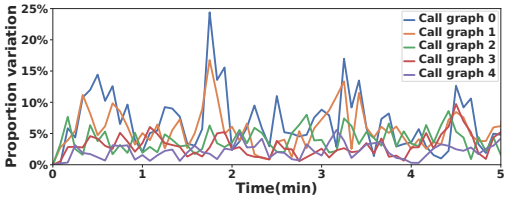


Figure 3: The call graph proportion variation of the largest microservice application in Alibaba Cloud.

the deep learning-driven methods to predict the end-to-end latency and estimate the resource allocation for different microservice stages, which can minimize the resource usage while ensuring the QoS target. These works only consider a unique dependency graph, which cannot handle the call graph dynamics of the microservice application. Moreover, Madu [34] predicted the load size for microservices based on time series prediction models with the consideration of dynamic call graphs into the loss function. However, it cannot handle the unpredictable dynamic load and call graph cases that commonly exist in production traces [2, 40].

3 Investigating Dynamic MS Applications

In this section, we first analyze the production trace in the current public cloud. Then, we introduce microservice benchmarks we make which have the dynamic call graph features. At last, we explore the challenges of the current microservice management systems in dealing with dynamics.

3.1 Dynamic Loads and Call Graphs

We analyze the open-sourced production-level microservice trace [2] that contains the microservice call dependencies across 3000+ applications in 12 hours to show the dynamics.

In the analysis, we record the loads of the microservices for every five seconds, and calculate the load variation. The load variation is defined to be the load changes in the adjacent samples. Figure 2 shows the cumulative distribution of the load variation of the microservices in the top 20 production-level microservice applications. The top-20 applications are selected according to the numbers of their queries. As observed, the load variation is from 10% to 40% for 70% of the samples. In the worst case, the load may increase by 2.3X.

From statistics, we find some of the top 20 microservice applications have a great many types of call graphs, with a maximum of 53761 types. All the microservices touched by a query form a call graph. Different queries may have the same call graph. Moreover, Figure 3 shows the call graph proportion variation over time of the top 5 call graphs in the largest microservice application. A call graph's proportion variation is defined to be its proportion changes in the adjacent samples. We can observe that the percentages of queries to the call graphs change dynamically with no oblivious pattern.

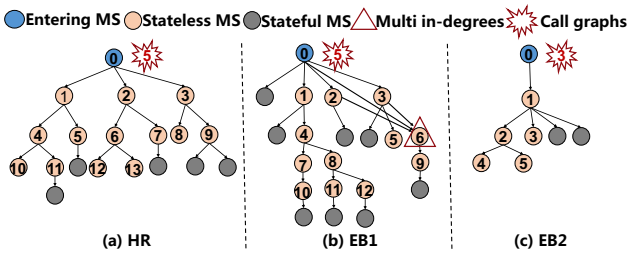


Figure 4: The dependency graphs of the benchmarks. Stateful microservices includes memcached and monogdb.

With the large number of call graphs and the unstable variation, it is non-trivial to profile the call graphs and their patterns, and define static optimal resource allocation beforehand.

3.2 The Investigation Benchmarks

While current microservice benchmarks do not support dynamic call graphs [24,42,46], and the traces [2] do not include the actual microservices (only the call traces), we build three benchmarks by integrating the call graph patterns of the trace and actual microservices in the benchmark suites.

Figure 4 shows the dependency graphs of the three benchmarks, and the number of call graphs. The *HR* benchmark is revised based on the popular HotelReservation benchmark [24]. It has five call graphs that respond to five types of user queries: nearest hotel search, highest rated hotel search, cheapest hotel recommend, comprehensive hotel recommend, and hotel reservation. The benchmarks *EB1* and *EB2* are created based on the dependency graphs of the top 2 applications with multiple call graphs in the trace. For *EB1* and *EB2*, similar to current benchmarks and related work [24,35,36,46], we use commonly-used workloads in microservices, i.e., Nearest Neighbor Searching [6], Word Stemming [13], Quick Sort [15], Float Calculation [43], and Page Rank [16] to be the stateless microservices.

3.3 The Long QoS Recovery Time

We show the QoS recovery time and 99%-ile latencies of the benchmarks with load and the call graph dynamics in this subsection. The QoS recovery time is the time needed to reduce the 99%-ile latency to be below a fixed latency target

Table 1: Experiment specifications

	Specifications
Hardware	Eight-node cluster, Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 256GB Memory Capacity, 25 MiB L3 Cache Size (20-way set associative)
Software	Ubuntu 20.04.2 LTS with kernel 5.11.0-34-generic Docker version 20.10.18, Kubernetes version v1.20.4

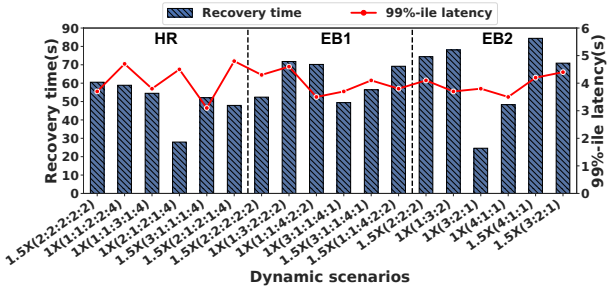


Figure 5: The QoS recovery time and 99%-ile latencies of benchmarks with ELIS.

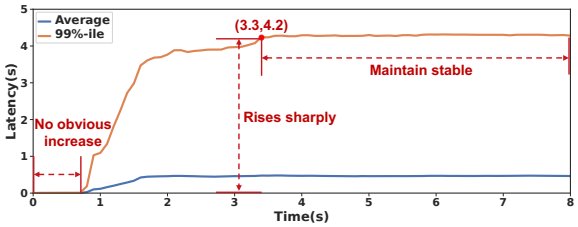


Figure 6: The exploration of latency monitoring intervals.

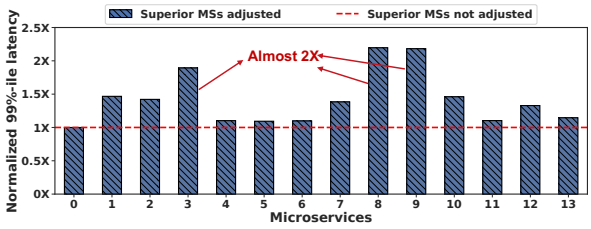


Figure 7: The 99%-ile latency of each microservice after its superior microservices have been allocated enough resources.

(e.g., 100ms) after microservice dynamics happen. Table 1 summarizes the detailed hardware and software configurations. ELIS [41] uses bayesian optimization to tune resource allocation for microservices. We use ELIS as the representative resource management system for microservices in this section. Other systems have similar results and we show them in the evaluation section.

In the experiments, each benchmark has 6 dynamic load and call graph scenarios. We run the experiments on three identical servers managed with Kubernetes [14]. We expand the evaluation on eight servers in Section 8. In each test, each microservice is allocated the hand-tuned enough resource, and the numbers of queries to different call graphs are the same. Figure 5 shows the QoS recovery time and 99%-ile latencies in all the test cases. The *x*-axis represents the dynamic scenarios. For instance, *1.5X(2:1:2:1:4)* means the load increases to 1.5*X*, and the proportions of the 5 call graphs change to 2/10, 1/10, 2/10, 1/10, and 4/10. As observed, the QoS recovery time ranges from 24.6 to 84.4 seconds, and the 99%-ile latencies range from 3.1 to 4.8 seconds, in all the test cases.

Both the two types of dynamics result in serious QoS violations and the QoS recovery time is long.

3.4 Causes of The Long Recovery Time

Our investigation shows the long QoS recovery time is caused by *long monitoring interval*, *execution blocking effect due to dynamcis*, and *slow query draining*.

3.4.1 Long Monitoring Interval

Current resource management systems monitor the realtime latencies of the microservices, and reallocate resources based on either heuristic methods [17, 27, 44] or machine learning based methods [23, 38, 45]. These systems often use seconds or tens of seconds to be the monitoring interval.

As an example, Figure 6 shows the 99%-ile latency and average latency of the benchmark *HR*, when we increase its load to 1.5X. The *x*-axis shows the time since we increase the load. As observed, the monitored 99%-ile latency has no obvious increase in the first second, even if the load already increases. The 99%-ile latency starts to increase sharply at about the first second, and becomes stable after 3.3 seconds. This is because the latencies of the newly arrived queries are not reported before they complete. In this case, the 99%-ile latency reported in the first second is actually the latency before the load actually increases.

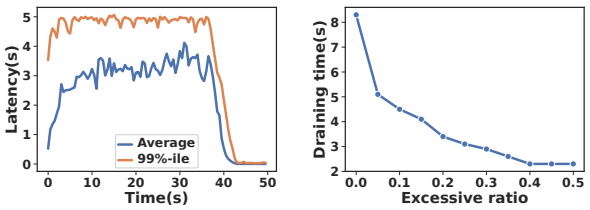
Long monitoring intervals are required for current systems that rely on the latencies of the microservices to adjust the resources. However, a great many of queries may already queue up at a microservice during the long monitoring interval.

3.4.2 Execution Blocking Effect

The second problem is that the monitored realtime load of a microservice may not be its actual "to-be-processed" load.

For instance, as shown in Figure 4(a), the load of microservice-3 may be blocked by microservice-0 if microservice-0 does not have enough computation resources. Similarly, the loads of microservice-8 and microservice-9 may also be blocked by microservice-3. There is more complex blocking effect, besides of the simple dependency relationship. The effect is referred to be *execution blocking effect* in this paper. We will analyze the effect in detail in Section 6.

Figure 7 shows the latencies of the microservices in the *HR* benchmark, when we allocated their superior microservices enough computation resources, normalized to their performance with the default resource allocation. The dynamic



(a) QoS recovery process. (b) Excessive resource ratio.

Figure 8: The results of queued query draining.

scenario of this experiment is *1.5X(2:1:2:1:4)*. As observed, the latencies of many microservices increase when their superior microservices get enough resources. This is because these microservices may get the real "to-be-processed" loads when the blocking effect is alleviated.

Due to the execution blocking effect, the monitored load of a microservice may be much smaller than its actual "to-be-processed" load. Current methods based on monitored load may not allocate enough resources for microservices, thus require to adjust the resource allocation for multiple times.

Similar to our conclusion, prior machine learning based and heuristic-based systems also notice that they require to adjust the resource allocation for multiple times [17, 22, 27, 38, 41]. For instance, the bayesian optimization based system, ELIS [41], needs to search for 4-15 samplings to find the final resource configuration for each microservice. Reinforcement learning-based system, FIRM [38], has to perform multiple incremental model updates, if the microservice application has dynamics. Each adjustment interval is also at least several seconds, incurring more queued queries.

Even if the optimal resource allocations can be determined directly, the long monitoring interval and the blocking effect already result in the long query queues. We also evaluate the optimal resource decision case in Section 8.

3.4.3 Slow Query Draining

The queued queries during the monitoring and the resource adjustment period can result in the long QoS recovery time.

As an example, Figure 8(a) shows the 99%-ile latency of the benchmark *HR* with ELIS, when we change the dynamic scenario to *1.5X(2:1:2:1:4)*. As observed, although appropriate resources are allocated to each microservice for its "to-be-processed" load, the 99%-ile latency gradually drops from time 39.6 seconds to 47.9 seconds instead of backing to normal immediately. This is because the resource allocation does not consider the queued queries at each microservice.

We further try to allocate excessive resources for microservices, and explore the impact of excessive ratio on queued query draining. The excessive ratio is the excessive resource allocation ratio for microservices after the resource adjustment process. Figure 8(b) shows the draining time under different excessive ratios. We can observe that the larger the excessive ratio, the shorter the draining time.

It is possible to reduce the queued query draining time through excessive resource allocation, so that can reduce the overall QoS recovery time. We should carefully determine the amount of excessive resources to ensure high resource efficiency, while minimizing the QoS recovery time.

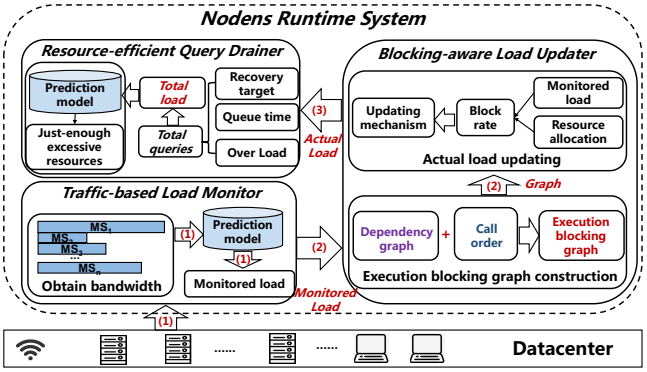


Figure 9: Design overview of Nodens.

4 Nodens Methodology

We design **Nodens** to enable the fast QoS recovery of dynamic microservice applications based on the above analysis.

Figure 9 shows the design overview of Nodens. It comprises a *traffic-based load monitor*, a *blocking-aware load updater*, and a *resource-efficient query drainer*. The monitor predicts the monitored load of a microservice based on the periodically obtained network bandwidth usage of the microservice. The load updater calculates the actual "to-be-processed" load of each microservice, based on the monitored loads of the microservices, the resource allocation of the microservices, and the execution blocking graph of the application. The query drainer allocates "just-enough" excessive resources for each microservice, to quickly drain the queued queries generated during the above process.

We use the network bandwidth usage to predict the load of a microservice for the short monitoring time. Section 5 shows that the incoming network bandwidth of a microservice is closely related to its realtime load. The monitored bandwidth is stable with the 1 second interval. With the short and stable monitoring interval, we can find the load variation quickly and tune the resource allocation as early as possible, reducing the number of queued queries at a microservice when the load increases.

The most challenging part is obtaining the actual "to-be-processed" load of each microservice, due to the execution blocking effect. We define an *execution blocking graph* for a microservice application. It reflects the blocking relationship among microservices, and is determined by the microservice dependency graph and microservice call order. A load updating mechanism is also required to capture the complex

realtime blocking actions due to the dynamics, based on the execution blocking graph (Section 6).

It is inevitable that some queries queue up at a microservice when its load increases, before the increase is noticed and the resource is adjusted. Without careful design, these queued queries result in serious QoS violations, or too much resource is allocated to handle the possible queued queries. The challenging part in the query drainer is to allocate just-enough excessive resources to ensure the QoS recovery time target while maintaining the resource efficiency (Section 7).

Specifically, Nodens manages the resource allocation of a microservice application in the following steps. 1) Nodens obtains the dependency graph and the possible call graphs of the application. Based on the obtained graphs, the *execution blocking graph* is built. 2) When serving the application, Nodens deploys a daemon process on each server to monitor the network usage of the microservices. 3) A server runs the load updater, and determines the resource allocation of each microservice with the query drainer. The load updater collects the bandwidth data of microservices on different servers, and calculates the actual "to-be-processed" load of every microservice. 4) The drainer updates the resource allocation accordingly, and sends back the allocation decision to each microservice. 5) The daemon process on each server then reallocates the resources based on the decision of the drainer.

Since the servers are in the same datacenter, we use gRPC [7] to collect the network usage of each microservice, and send back the allocation decision. The transfer latency is less than 5ms in our experiments. Nodens does not need to modify the source code of microservice applications, and can be implemented as a plug-in based on Kubernetes [14]. Moreover, Nodens does not focus on microservice deployment among distributed servers, and the initial deployment is determined by Kubernetes's random scheduling strategy.

Similar to prior works [29, 37, 41, 45], Nodens uses Linux cgroups [4] to adjust CPU resources, which can complete within 1ms. VPA [10] in Kubernetes also supports in-place pod vertical scaling with low overhead. After each allocation decision, if there are no resources available on some servers in the first place for vertical scale-up, Nodens utilizes the resource recycling idea [41] to deal with. Nodens will first recycle the resources from over-provisioned microservices on these servers, and then allocate them to microservices requiring scaling up. If some servers still lack sufficient resources for their deployed microservices after resource recycling, Nodens adopts current load balancing strategies [22, 33, 41] to migrate some microservices from busy servers to idle servers.

5 Traffic-based Load Monitor

5.1 The Speedup and Predictability

As discussed in Section 3.4.1, the latency monitoring can result in long resource adjustment time. Therefore, we use the upper network bandwidth usage for resource adjustment to reflect the load change of microservices. For microservice-6 of the *EB1* benchmark shown in Figure 4, its upper network bandwidth usage is the data communication amount per second from microservices 0, 2, and 3.

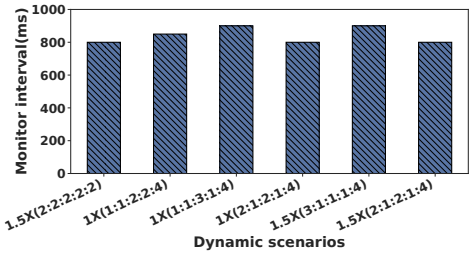


Figure 10: Network traffic monitoring intervals.

To explore the required interval of network traffic monitoring, we conduct 6 experiments on the *HR* benchmark, whose dynamic scenarios are the same as Section 3.3. For each dynamic scenario, we first run the benchmark and monitor all microservices for 10 seconds to obtain their stable network bandwidth usage (Mbits/s) as the baseline values. Then, we run the benchmark again and gradually increase the intervals (starting from 50ms) to find the minimum interval that may get stable monitoring data. We consider the monitoring data to be stable when the error between obtained microservices' bandwidth usage and corresponding baseline values are within 5%. Figure 10 shows that the obtained minimum monitoring intervals are less than 1000ms. Compared with the latency monitoring interval (3300ms) we test in Section 3.4.1, the required interval of monitoring network is 3X shorter.

In addition, we find the upper network bandwidth usage has a typical linear relationship with the load size (i.e., queries per second, QPS) for all microservices, and the relationship between load size and CPU core demand is the same. For a microservice application, we profile each call graph at 10 sets of loads (evenly from 0 to the peak supported load), and obtain the performance samples (i.e., the load, upper network bandwidth usage, and CPU core demand) of all microservices. The profiling can be done automatically and online. For long-running applications, call graphs can be known from history. Otherwise, we can trace the call graph and profile the new call graphs online. We use the profiled performance samples to train the linear models for microservices. We then use the performance samples at 10 other different sets of loads as the test dataset. Predicting the load size through the network bandwidth, and predicting the CPU core demand through the load size, the prediction accuracies are 97.0% and 97.9% on average for the 3 benchmarks, respectively. So, we can accurately predict the load size of each microservice through its upper network bandwidth, and further predict its CPU core demand. As prior works have shown that microservices are basically sensitive to CPU resources [31, 34, 45], Nodens primarily focuses on CPU core allocation. From our observations, the

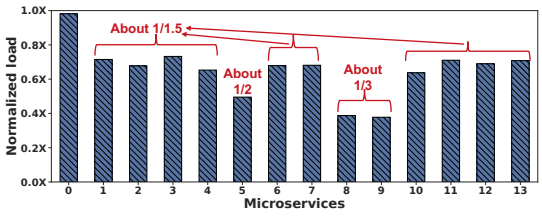


Figure 11: The monitored loads normalized to the actual loads of microservices.

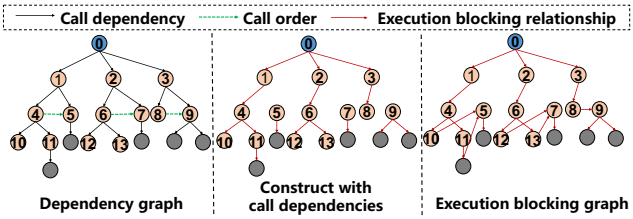


Figure 12: The dependency graph, and execution blocking graph of the *HR* benchmark.

memory usage of microservices is steady, and we pre-allocate enough memory capacity for microservices.

Compared with the ML-based methods [38,41], we can use these models to predict the CPU core demand from network bandwidth usage, whose overhead is usually less than 1ms.

5.2 Network Traffic Monitoring Methods

As the analysis of production microservice applications [31, 32], most of them have tree-like dependency graphs, while a few of them show graph-like structures. The in-degrees of some microservices in a graph-like structure are larger than 1.

For the tree-like dependency graph, we obtain the receive and transmit bytes of microservices' corresponding network interface during the monitoring interval by reading Linux file */proc/net/dev*, and then calculate the upper network bandwidth one by one based on the tree structure. The overhead of this method is less than 15ms, which includes reading the Linux file and calculations. For the graph-like dependency graph, we use Libpcap [12] to obtain network traffic between each pair of microservices directly, and then calculate the upper network bandwidth for each microservice. This method completes in 30ms. After obtaining the upper network bandwidth usage of microservices, the monitor uses linear regression models to predict the monitored loads of them as this module's output.

6 Blocking-aware Load Updater

6.1 Execution Blocking on Monitored Load

Since the effect of execution blocking under dynamics, the monitored loads may not equal to the actual "to-be-processed" loads of microservices. In this subsection, we explore the effect of execution blocking on monitored loads with the *HR* benchmark. We conduct an experiment from the initial state of *1X(2:2:2:2:2)* to the dynamic state of *1.5X(3:1:1:1:4)*. Figure 11 shows the normalized monitored loads to the actual loads of microservices. The left part of Figure 12 shows the dependency graph of *HR*.

As observed, since the load changes from 1X to 1.5X, we can find most microservices' monitored loads are about $\frac{1}{1.5}$ of their actual loads, as their superior microservices can only handle 1X load with current resource allocation. Microservices-8 and 9' monitored load is about $\frac{1}{3}$ of their actual loads, as the proportion of their located call graph also increases two times. Above observations prove that execution blocking effect can be caused by the **call dependencies among microservices**.

Moreover, microserivce-5's monitored load is about $\frac{1}{2}$ of its actual load, but not $\frac{1}{1.5}$. After looking into the dependency graph, we find since microservice-4 and microservice-5 are called by microservice-1 in a fixed order, the microservices in the subtree of microservice-4 can be the execution blocking microservices of microservice-5. In detail, since the load changes and the proportion of microservice-4's located call graphs also changes, microservice-4's subtree blocks $\frac{1}{2}$ of the actual load, which cannot be passed to microservice-5. Above observations prove that the execution blocking effect can be caused by the **call order among microservices**.

We give a simple example for Figure 12. Suppose queries pass through part of the microservices in following orders: 1) log in (microservice-3), 2) authentication (microservice-8), and 3) reservation (microservice-9). Suppose microservice-3's loads increase to 1500 queries per second but it only has just-enough resources to handle loads of 1000. At this point, the loads to its downstream microservices are blocked at 1000, which is 1/1.5 of 1500 (microservices-8 and 9's actual loads).

If we adjust resources only based on monitored loads, we need to adjust resources for microservices multiple times to deal with the execution blocking, which can greatly increase the QoS violation time. Therefore, a reasonable method is to combine the microservice dependency graph, monitored load, and resource allocation to update the actual loads of microservices in advance. During this process, we need to primarily consider the execution blocking effect caused by **call dependencies** and **call order** among microservices.

6.2 Execution Blocking Graph

Based on the observations in Section 6.1, we construct the **Execution Blocking Graph** for the microservice application.

Figure 13 shows the execution blocking graph construction based on the microservice dependency graph. The microservice dependency graph is obtained by using tracing tools (e.g., Jaeger [11]) after running the application online for one minute. Firstly, for the microservice in the dependency graph that has multiple in-degrees, we transform the subtree with it

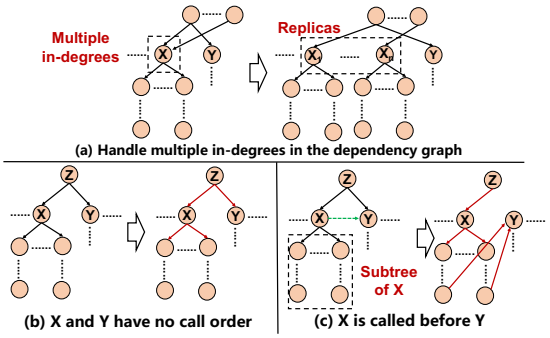


Figure 13: Execution blocking graph construction.

to multiple identical replicas, to maintain the tree structure of the dependency graph, as shown in Figure 13(a). The replicated overhead is low, because few microservices have multiple in-degrees in production microservice applications [31].

We then construct the execution blocking relationship for each sub structure. On the one hand, as shown in Figure 13(b), microservices X and Y have no fixed call order from their common superior microservice. In this case, the execution blocking relationship is equal to the call dependencies among microservices. On the other hand, as shown in Figure 13(c), X is called before Y by their common superior microservice. In this case, the execution blocking microservices of Y are the ones at the end of the execution blocking subtree with root X. The number of this kind of microservices can be one or more. We take an example for better explanations. Suppose X is the "authentication" microservice, Y is the "reservation" microservice, and Z is the "log in" microservice. As X is called before Y by Z, the resource insufficiency of X can block Y. By contrast, if X and Y are called by Z asynchronously, X cannot block Y.

Breaking down the microservice dependency graph into multiple sub structures, we iteratively build the execution blocking relationship from the root microservice. Then, we can obtain the execution blocking graph for the microservice application. Nodens does not need to replicate microservices with multiple in-degrees in the execution blocking graph.

We use the example of *HR* benchmark in Figure 12 to further explain the whole construction process. As microservices 1, 2, and 3 have no fixed call order, their execution blocking relationship is equal to their call dependencies. Some other microservices have similar construction, and the middle results are shown in the middle part of Figure 12. Moreover, microservice-5 is called after microservice-4 by their superior microservice-1. Therefore, its execution blocking microservices are the microservices at the end of the execution blocking subtree of microservice-4. Microservices 7 and 9 have similar construction process with considerations of the call order. The final execution blocking graph is shown in the right part of Figure 12.

The execution blocking graph differs from the execution graph in 2 ways. First, actually an execution graph is similar to a call graph, while the execution blocking graph is constructed once from the dependency graph and it captures all the possible blocking relationship. Second, we define the node and edge weights in the execution blocking graph for updating the actual loads for microservices (Section 6.3), while execution graphs do not have such information.

6.3 Actual Load Updating Mechanism

Based on the execution blocking graph, we then introduce the actual load updating mechanism.

We define the triple to record current state of each microservice i as $(MonitoredLoad_i, ActualLoad_i, HandleLoad_i)$. The $MonitoredLoad_i$ is obtained from the traffic-based load monitor. The $ActualLoad_i$ will be updated by the mechanism for each microservice, and is equal to the $MonitoredLoad_i$ at the beginning. The $HandleLoad_i$ represents the load that can be handled for each microservice. It is predicted by using the linear regression model from its corresponding microservice's resource allocation. For microservices i and j in the execution blocking graph, the edge weight EW_{ij} is defined as the load passing from i to j. The EW_{ij} is equal to the monitored load from i to j at the beginning, and will be updated during the updating mechanism.

Since microservices may block the load of their downstream microservices, we then define the blocking rate of the microservice j as:

$$rate_j = max(\frac{ActualLoad_j}{min(HandleLoad_j, MonitoredLoad_j)}, 1) \quad (1)$$

In this formula, the blocking rate is the $ActualLoad$ dividing the minimum of the $HandleLoad$ and $MonitoredLoad$, as the former may be larger than the latter since higher-level blocking. Moreover, as the loads of some microservices may decrease under dynamics, the blocking rate may be smaller than 1, and we set $rate_j = 1$ for these cases.

We mainly adopt the Breadth-First-Search (BFS) algorithm based on the execution blocking graph to calculate the blocking rate and update the actual loads of microservices layer by layer. Algorithm 1 shows the mechanism. We first initialize the load triple for microservices, the execution blocking graph, and a queue for the BFS process. We then put the root microservice into the queue, and then come into the major process of actual load updating. During this process, we first calculate the blocking rate of the head microservice j of the queue based on Eq.(1), whose $ActualLoad$ has been updated correctly, as shown in the lines of 6-8. Then, we start to handle the downstream microservices of j, as shown in the lines of 9-14. For each downstream microservice k, we first update the actual load from j to k with the blocking rate of j. If k's all entry edges are all updated, we then put it into the queue to follow the process of the BFS algorithm. The updating process is ended when the queue is empty, which represents the

Algorithm 1: Actual Load Updating Mechanism

```
Initialize (MonitoredLoad_i, ActualLoad_i, HandleLoad_i)
Initialize execution blocking graph EBG with edge weights EW_ij
Initialize a queue q for the BFS process
q.put(EBG.root)
while q ≠ ∅ do
    j = q.get()
    ActualLoad_j = Σ_{i→j} EW_ij
    rate_j = max(ActualLoad_j / min(HandleLoad_j, MonitoredLoad_j), 1)
    for each downstream microservice k of j do
        EW_jk = EW_jk × rate_j
        if all entry edges of k are updated then
            q.put(k)
        end if
    end for
end while
return ActualLoads for all microserivces
```

actual loads of all microservices have been updated. Lastly, we return the actual loads of microservices.

7 Resource-efficient Query Drainer

Although the resource adjustment time of Nodens is greatly decreased compared to the latency based resource adjustment methods, the queued query draining is non-negligible, as just-enough resources can lead to the queued queries being unable to be drained for a long time. As discussed in Section 3.4.3, the larger amount of excessive resources can accelerate the draining process, but obviously sacrifice the resource efficiency. We define the QoS recovery time as the time needed to reduce the 99%-ile latency to be below a fixed latency target (e.g., 100ms) after microservice dynamics happen. Same to prior work [21, 38, 41, 45], the QoS is often defined to be latency. Nodens can also support other QoS definitions (e.g., throughput) through simple adaption. We set the QoS recovery time target for the microservice application, e.g., the QoS recovery time is within 3 seconds after microservice dynamics happen. Therefore, the excessive resource allocation of the microservice application can be described as *minimizing the excessive resource allocation on the premise of guaranteeing the recovery time target*.

To allocate just-enough excessive resources for each microservice, we try to drain the queued queries for each microservice exactly within the recovery time target. Therefore, our goal is to calculate the total queries to be processed for each microservice during the residual recovery time, and then allocate just-enough excessive resources correspondingly.

We first calculate the overload of each microservice i during the resource adjustment process as:

$$OverLoad_i = ActualLoad_i - min(MonitoredLoad_i, HandleLoad_i) \quad (2)$$

where the $ActualLoad_i$, $MonitoredLoad_i$, and $HandleLoad_i$ have the same definition to Section 6. For some microservices, their actual loads may be less than or equal to their monitored and handle loads under microservice dynamics. For these cases, we set $OverLoad_i = 0$ for them.

After calculating the $OverLoad_i$, we can further calculate the total queries to be processed for each microservice during the residual recovery time as:

$$TotQuery_i = OverLoad_i \times T_i + ActualLoad_i \times (QT - T_i) \quad (3)$$

where T_i is the resource adjustment time which causes query queuing, and QT is the recovery time target. In this formula, the first item represents the total amount of queued queries, while the second represents the total amount of normal queries that need to be processed during the residual recovery time.

At last, we can calculate the total load (i.e., queries per second) that needs to be handled during the residual QoS recovery time for each microservice i as:

$$TotLoad_i = \frac{TotQuery_i}{(QT - T_i)} \quad (4)$$

Obtaining the $TotLoad$, we can use the linear model mentioned in Section 5.1 to predict the total CPU core demand of each microservice, and then tune the allocated resources accordingly. The total CPU core demand includes the just-enough resources under the corresponding dynamic scenario and excessive resources for queued query draining. After QoS is recovered, the excessive resources will be recycled to maintain high resource efficiency.

8 Evaluation of Nodens

In this section, we first evaluate the performance of Nodens in recovering the QoS while achieving resource efficiency. Then, we show the effectiveness of the blocking-aware load updater, and the resource-efficient query drainer.

8.1 Evaluation Setup

Table 1 already shows the configurations of the experimental platform. We evaluate all the three benchmarks *HR*, *EB1*, and *EB2* on the eight-node cluster in the experiments. For each benchmark, we evaluate Nodens with six dynamic scenarios, including load dynamic, call graph dynamic, and the mix of the two types of dynamics.

We compare Nodens with two state-of-the-art microservice management systems FIRM [38] and ELIS [41]. FIRM monitors the latencies of microservices periodically, identifies the critical path and critical microservices, and increases their resources to the optimal resource allocations using reinforcement learning. ELIS first recycles the over-provisioned resources of non-critical microservices before reallocating the resources. It uses bayesian optimization to reallocate resources. For FIRM and ELIS, the latency monitoring periods

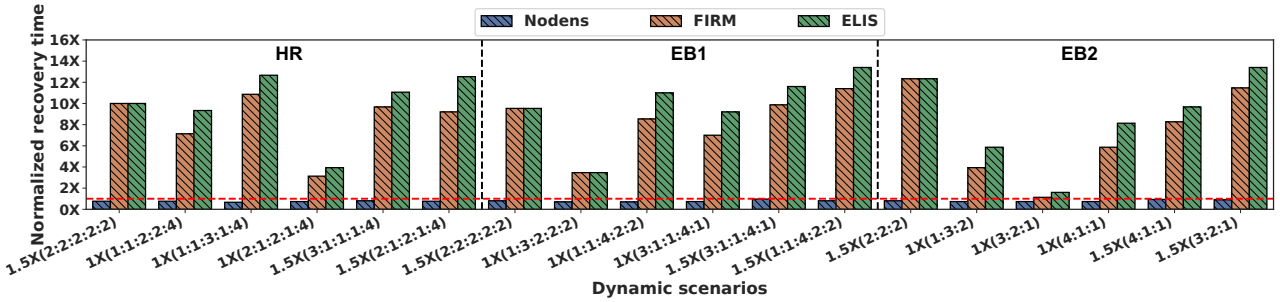


Figure 14: The normalized QoS recovery time relative to the recovery time target of benchmarks with Nodens, FIRM, and ELIS.

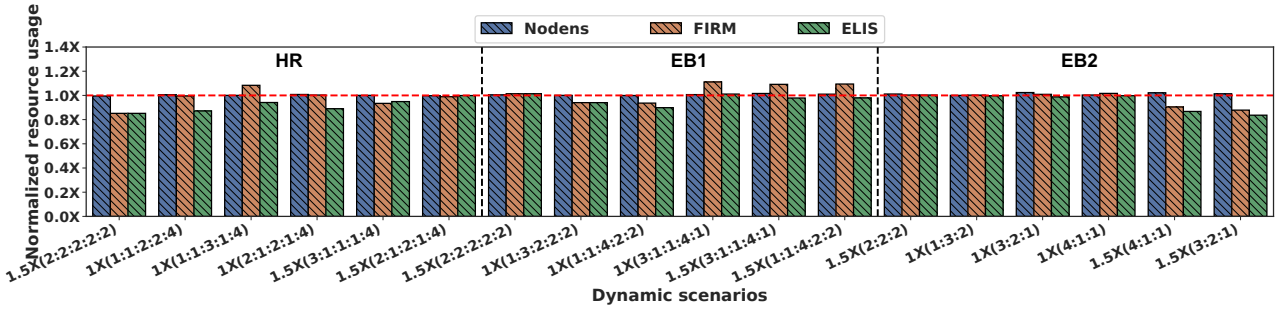


Figure 15: The normalized resource usage relative to the just-enough resources of benchmarks with Nodens, FIRM, and ELIS.

are set to be the minimum time with which the latency is stabilized (i.e., the subsequent latency increase is less than 5%). In Nodens, we use 1 second to be the network traffic monitoring interval as analyzed in Figure 10.

In our experiments, we already optimize FIRM and ELIS through offline profiling. While ML-based resource adjustment requires multiple iterations to find the optimal resource configuration, we optimize them to be able to directly allocate the optimal resources for microservices. Moreover, we also give each microservice excessive resources according to the recommendation of Nodens's query drainer. The native FIRM and ELIS perform worse than the ones we used here.

In the following experiments, we use 3 seconds to be the recovery time target. In other words, Nodens will adjust the resource allocation, in order to make sure that the QoS violation is eliminated in 3 seconds.

8.2 QoS Recovery and Resource Efficiency

In all test cases, microservices are initially allocated the just-enough resources when the load is 1X and the percentages of queries that go to different call graphs are identical. Then, we change the loads of the entire benchmark and the percentages of queries go to different call graphs, and evaluate the performance of Nodens in recovering the QoS before the given QoS recovery target.

Figure 14 shows the QoS recovery time of all the $3 \times 6 = 18$ test cases with FIRM, ELIS, and Nodens, respectively. The time is normalized to the recovery time target (3 seconds). In the figure, "1.5X(2:2:2:2:2)" represents the case that the application's load increases to 1.5X, and the percentages of queries to the five call graphs are identical. As observed from the figure, Nodens successfully eliminates the QoS violation in the given recovery time target. By contrast, the QoS recovery time with FIRM and ELIS is 7.9X and 9.4X of the recovery time target, and 10.2X and 12.1X of Nodens's.

Nodens has shorter QoS recovery time because it has shorter but stable load monitoring interval, and calculates the actual "to-be-processed" load of each microservice. It is able to reduce the queued queries during the monitoring interval, and allocate enough resources for each microservice beforehand. We can also find that the QoS recovery time is longer with ELIS than with FIRM. This is because ELIS first recycles the over-provisioned resources, which can spend some extra time. Moreover, the QoS recovery time is also short with ELIS and FIRM in some cases (e.g., the scenario *1X(3:2:1) with EB2*). It happens when there are only a few microservices' resources are insufficient.

Figure 15 shows the corresponding total resource usage (*cores* $\times$ *hours*) of the test cases during the QoS recovery process. The resource usage is normalized to the case that all the microservices have "just-enough" resources for the new load since the dynamic happens. We use the longest QoS recovery time (i.e., ELIS's) to calculate the total resource usage for the fair comparison. As observed, Nodens uses 1.5% and 6.1% more resources on average than FIRM and ELIS, respectively.

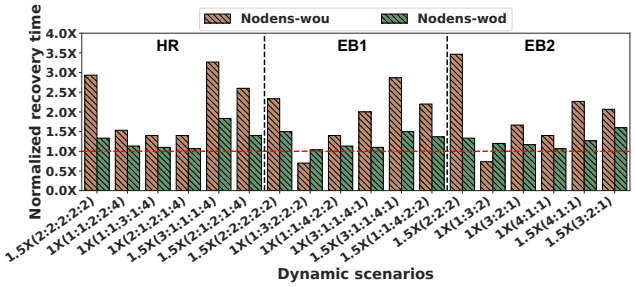


Figure 16: The QoS recovery time with Nodens-wou and Nodens-wod normalized to the recovery time target.

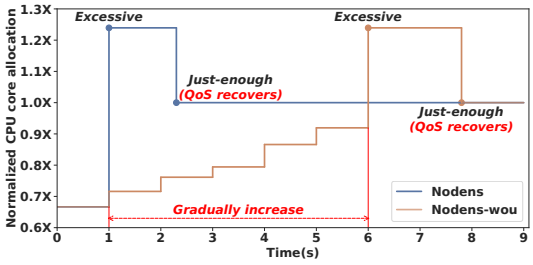


Figure 17: The CPU core allocation timeline of Nodens and Nodens-wou in an example dynamic scenario.

FIRM also uses more resources than ELIS. This is because FIRM only increases the resources of critical microservices without recycling the over-provisioned resource.

Therefore, Nodens is resource efficient while realizing the fast QoS recovery.

8.3 Effectiveness of the Load Updater

In this experiment, we show the performance of *Nodens-wou*, a variant of Nodens that disables the blocking-aware load updater. With Nodens-wou, the query drainer still allocates excessive resources for the microservices.

The orange bars of Figure 16 show the QoS recovery time of all the test cases with Nodens-wou normalized to the QoS recovery time target. As observed, Nodens-wou recovers the QoS before the recovery time target in only two cases. Compared with Nodens, Nodens-wou requires 2.6X time on average to recover the QoS.

As an example, Figure 17 shows the normalized resource allocation timeline of Nodens and Nodens-wou in the test case *1.5X(2:1:2:1:4)* of the *HR* benchmark. Other test cases show similar conclusions. As shown in the figure, Nodens allocates excessive resources to the microservices at an early time, and returns to the "just-enough" resource allocation once the QoS violation is eliminated. On the contrary, Nodens-wou gradually increases the resource allocated to the microservices after each monitoring interval. This is because the execution blocking effect makes Nodens-wou cannot obtain the actual "to-be-processed" loads of the microservices. For a microservice, its load pressure is released layer by layer from its superior microservices.

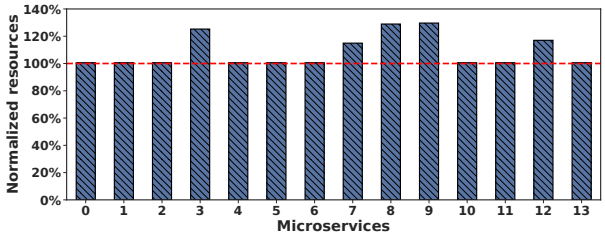


Figure 18: The resource allocation of microservices normalized to the just-enough resources for an example experiment.

The blocking-aware load updater is necessary for Nodens. It avoids the execution blocking effect by updating the actual "to-be-processed" loads of microservices in advance.

8.4 Effectiveness of the Query Drainer

In this experiment, we show the performance of *Nodens-wod*, a variant of Nodens that disables the query drainer. The blocking-aware load updater still works in Nodens-wod.

The green bars of Figure 16 show the QoS recovery time of all the test cases with Nodens-wod normalized to the QoS recovery time target. As observed, Nodens-wod fails to recover the QoS before the recovery time target in all the cases. Compared with Nodens, Nodens-wod requires 1.6X time on average to recover the QoS. This is because the queued queries generated during the resource adjustment cannot be drained up quickly without the excessive resources allocated by the query drainer. Moreover, we can find that Nodens-wod performs better than Nodens-wou in most cases. This is because Nodens-wod can eliminate the execution blocking effect which is the most important influence factor that causes long-time QoS violations.

As an example, Figure 18 shows the actual resources allocated of microservices normalized to the just-enough resources in the test case *1X(1:1:3:1:4)* of the *HR* benchmark with Nodens. As observed, Nodens allocates excessive resources for 5 of the 14 microservices in the test case. Specifically, microservices 3, 8, and 9 belong to the same call graph and are affected by the same degree of execution blocking, so they have almost the same ratio of excessive resources. Microservices 7 and 12 belong to another call graph, and their resource shortage is smaller than the previous 3 microservices, so Nodens's drainer allocates them less excessive resources.

8.5 The Impacts of Dynamics

Since there are load dynamics, call graph dynamics, and load+call graph dynamics when serving a microservice application, we evaluate their impacts on the query drainer.

Figure 19 shows the ratio of total excessive resource allocation for all dynamic scenarios of the 3 benchmarks with

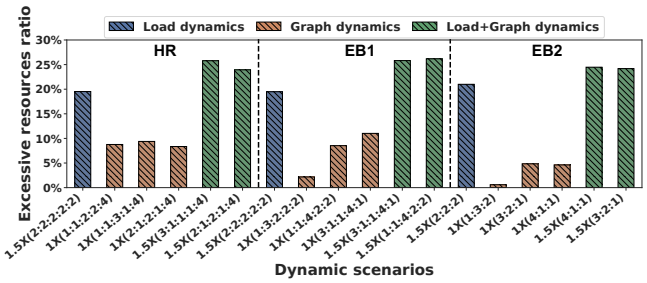


Figure 19: The ratio of total excessive resource allocation for all dynamic scenarios of the three benchmarks.

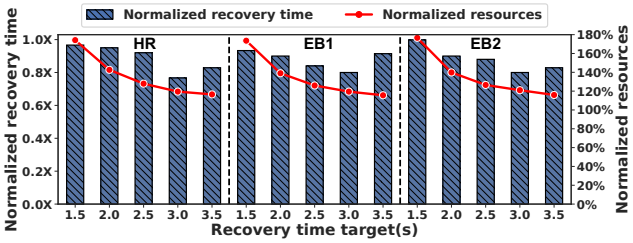


Figure 20: The QoS recovery time and resource allocation under different recovery time targets.

Nodens. As observed, the excessive resource ratio is smallest with only call graph dynamics, larger with only load dynamics, and the largest with both load and call graph dynamics.

This is because dynamic call graph scenarios only cause a few microservices' resources to be insufficient and dynamic load scenarios cause more, while the simultaneous dynamic load and call graph scenarios cause the most. As the resource shortage of more microservices can cause more queries queued, Nodens' drainer will allocate more excessive resources for these scenarios.

8.6 Handling Different Recovery Time Targets

In this experiment, we show Nodens's performance in handling different recovery time targets. We use one dynamic case of each benchmark to conduct the experiment, i.e., 1.5X load with identical call graph percentages.

Figure 20 shows the QoS recovery time and the actual resource allocation for all the cases. The QoS recovery time and the actual resource allocation are normalized to the recovery time target and just-enough resources in each case, respectively. As observed, Nodens successfully eliminates the QoS violation in different given recovery time targets for all the cases. Moreover, Nodens allocates more/fewer resources for the case with the smaller/larger recovery time target, which proves Nodens's resource efficiency.

Therefore, Nodens can ensure different QoS recovery time targets, while maintaining resource efficiency.

8.7 Overhead of Nodens

Offline Overhead. To train the prediction models for CPU core allocation, Nodens needs to profile the bandwidth and performance data for different microservices at different loads in advance. The offline profile time is about 25 minutes for each benchmark. The offline training time of the linear regression models for each benchmark is less than 150 ms.

Online Overhead. After deploying Nodens online, the execution time of the load monitor to get network traffic is less than 30ms. Moreover, the execution time for the load updater and query drainer are both less than 5ms. The prediction time and CPU core allocation time are all less than 1ms. The data transfer latency between servers is less than 5ms. Therefore, the total online overhead is about 50ms. The overhead is acceptable as it is far less than the monitoring interval in our experiments (i.e., 1 second).

We also evaluate Nodens's overhead using a simulated large-scale application with 200 microservices and 10 call graphs based on production-level microservice traces [31]. The online overhead is 126.6ms (59.4ms, 31.6ms, and 35.6ms for the network monitor, load updater, and query drainer, respectively), the offline profiling overhead is 50 minutes, and models can be trained in 2s.

9 Conclusion

In this paper, we propose Nodens to enable fast QoS recovery of dynamic microservice applications, while maintaining the efficiency of resource usage. Nodens's traffic-based load monitor predicts the monitored loads for microservices based on their network bandwidth usage. Nodens's blocking-aware load updater calculates the actual "to-be-processed" loads of microservices based on the execution blocking graph. It can eliminate the execution blocking effect, so that can reduce the total resource adjustment time. The query drainer allocates excessive resources for microservices to drain the queued queries, ensuring the QoS recovery time target. We have implemented Nodens and the experimental results show that, compared to the state-of-the-art microservice management systems, Nodens reduces the QoS recovery time by 12.1X.

Acknowledgment

We sincerely thank Mohammad Shahrad and our anonymous reviewers, for their helpful comments and suggestions. This work is partially sponsored by the National Natural Science Foundation of China (62232011, 62022057, 61832006), and Shanghai international science and technology collaboration project (21510713600). Quan Chen and Minyi Guo are the corresponding authors.

References

[1] Adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[2] Alibaba microservice cluster trace v2021. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021.

[3] Case studies of Sina Weibo. https://www.alibabacloud.com/help/en/function-compute/latest/sina-weibo.

[4] cgroups(7) — linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[5] Design Twitter — Microservices Architecture of Twitter Service. https://thinksoftware.medium.com/design-twitter-microservices-architecture-of-twitter-service-996ddd68e1ca.

[6] Golearn. https://github.com/sjwhitworth/golearn.

[7] gRPC. https://grpc.io/.

[8] Huawei Cloud Geo-distributed VM Traces. https://github.com/shijiuchen/HuaweiCloud-GeoVMTraces.

[9] Implementing microservices on AWS. https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html.

[10] In-place update of pod resources. https://github.com/kubernetes/enhancements/issues/1287.

[11] Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[12] Libpcap 1.x.y by the tcpdump group. https://github.com/the-tcpdump-group/libpcap.

[13] Porter Stemmer for Go. https://github.com/a2800276/porter.

[14] Production-grade container orchestration. kubernetes.io.

[15] QuickSort - Go Packages. https://pkg.go.dev/github.com/TannerGabriel/learning-go/algorithms/sorting/QuickSort.

[16] Weighted PageRank implementation in Go. https://github.com/alixaxel/pagerank.

[17] Ataollah Fatahi Baarzi and George Kesidis. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 427–441, 2021.

[18] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. DeepRest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 181–198, 2022.

[19] 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*, pages 153–167, 2017.

[20] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 183–198, 2022.

[21] Kaihua Fu, Jiuchen Shi, Quan Chen, Ningxin Zheng, Wei Zhang, Deze Zeng, and Minyi Guo. QoS-aware irregular collaborative inference for improving throughput of dnn services. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 993–1006. IEEE Computer Society, 2022.

[22] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, and Minyi Guo. Adaptive resource efficient microservice deployment in cloud-edge continuum. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1825–1840, 2021.

[23] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[24] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 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*, pages 3–18, 2019.

[25] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer:

Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[26] Alim Ul Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004. IEEE, 2019.

[27] Md Rajib Hossen, Mohammad A Islam, and Kishwar Ahmed. Practical efficient microservice autoscaling with QoS assurance. In *In Proceedings of HPDC*, 2022.

[28] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408. IEEE, 2020.

[29] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, 2022.

[30] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, 2022.

[31] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[32] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022.

[33] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient resource management for shared microservices with SLA guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 62–77, 2022.

[34] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 355–369, 2022.

[35] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F Wenisch. Parslo: A gradient descent-based approach for near-optimal partial SLO allotment in microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 442–457, 2021.

[36] Amirhossein Mirhosseini, Brendan L West, Geoffrey W Blake, and Thomas F Wenisch. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 207–219. IEEE, 2020.

[37] Pu Pang, Quan Chen, Deze Zeng, and Minyi Guo. Adaptive preference-aware co-location for improving resource utilization of power constrained datacenters. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):441–456, 2020.

[38] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825, 2020.

[39] Jiu-Chen Shi, Xiao-Qing Cai, Wen-Li Zheng, Quan Chen, De-Ze Zeng, Tatsuhiro Tsuchiya, and Min-Yi Guo. Reliability and incentive of performance assessment for decentralized clouds. *Journal of Computer Science and Technology*, 37(5):1176–1199, 2022.

[40] Jiuchen Shi, Kaihua Fu, Quan Chen, Changpeng Yang, Pengfei Huang, Mosong Zhou, Jieru Zhao, Chen Chen, and Minyi Guo. Characterizing and orchestrating VM reservation in geo-distributed clouds to improve the resource efficiency. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 94–109, 2022.

[41] Jiuchen Shi, Jiawen Wang, Kaihua Fu, Quan Chen, Deze Zeng, and Minyi Guo. QoS-awareness of microservices with excessive loads via inter-datacenter scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 324–334. IEEE, 2022.

[42] Akshitha Sriraman and Thomas F Wenisch. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[43] Tetsuro Yamamoto. Historical developments in convergence analysis for Newton's and Newton-like methods. *Journal of Computational and Applied Mathematics*, 124(1-2):1–23, 2000.

[44] Wei Zhang, Quan Chen, Kaihua Fu, Ningxin Zheng, Zhiyi Huang, Jingwen Leng, and Minyi Guo. Astraea: towards QoS-aware and resource-efficient multi-stage GPU services. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 570–582, 2022.

[45] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[46] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows

Darby Huye
Tufts University, Meta

Yuri Shkuro
Meta

Raja R. Sambasivan
Tufts University

Abstract

The microservice architecture is a novel paradigm for building and operating distributed applications in many organizations. This paradigm changes many aspects of how distributed applications are built, managed, and operated in contrast to monolithic applications. It introduces new challenges to solve and requires changing assumptions about previously well-known ones. But, today, the characteristics of large-scale microservice architectures are invisible outside their organizations, depressing opportunities for research. Recent studies provide only partial glimpses and represent only single design points. This paper enriches our understanding of large-scale microservices by characterizing Meta's microservice architecture. It focuses on previously unreported (or underreported) aspects important to developing and researching tools that use the microservice topology or traces of request workflows. We find that the topology is extremely heterogeneous, is in constant flux, and includes software entities that do not cleanly fit in the microservice architecture. Request workflows are highly dynamic, but local properties can be predicted using service and endpoint names. We quantify the impact of obfuscating factors in microservice measurement and conclude with implications for tools and future-work opportunities.

1 Introduction

Microservice architectures are the de-facto method for building distributed systems in large-scale organizations [8, 13]. The basic tenants of this architectural style are well-known—monolithic applications are decomposed into smaller software services that communicate with one another over well-defined APIs, facilitating independence of different development teams, increased deployment velocity, and fine-grained scaling [11, 22]. But, outside of this basic understanding, there is a lack of clarity about industrial microservice architectures' design choices and their resulting characteristics. This ambiguity curtails the impact of microservices research. It is impossible to identify the microservice designs to which improvements suggested in the literature are best suited or which assumptions about microservices' characteristics are valid.

There has been a plethora of research seeking to improve the community's understanding of microservices. Many are qualitative, focusing on reasons for deploying microservice architectures [18, 24, 39], methods for decomposing monolithic applications to microservice architectures with many smaller services [9, 17, 36], and difficulties introduced by microservice architectures [39]. Though useful, they do not provide quantitative data about different organizations' microservice architectures, such as (but not limited to) their scale, topologies, or communication methods, all of which are critical to inform future research.

The community has also created many open-source testbeds built with the microservices design philosophy [1, 13, 46]. But, their scale and complexity do not match that of large-scale organizations' microservice architectures. Past research has shown that these testbeds exhibit much simpler behaviors than industrial architectures [31]. As a result, quantitative data about microservices obtained from these testbeds are not representative of industrial microservice architectures where the microservice architectural style is perhaps most valuable. This is concerning due to the number of research papers that rely on these testbeds [12, 14, 23, 25, 37, 38, 40, 43, 44]. For example, Sage [12] assumes synchronous RPCs. Tprof's [14] layer 4 grouping assumes non-combinatorial explosion when grouping requests by visited services' execution order. Both assumptions are invalid at Meta.

Recent publications from other large cloud companies provide quantitative data about their microservice architectures [20, 41]. But, they represent only partial views of single design points. Additional quantitative studies—both confirming existing findings and focusing on unexplored dimensions— are needed to enrich the community's understanding of large-scale microservices. We envision that these studies will collectively inform robust assumptions for use in microservice research and development.

We present a top-down analysis of Meta's microservice architecture, starting from its service-level topology and descending into individual request workflows. (Request workflows describe the order and timing of services visited by requests when executing.) Our focus is on underreported characteristics of microservice architectures important for developing microservice tools and artificially modeling microservice topologies. Specifically, we describe growth and churn of the microservice topology (to inform tools that learn models of the topology [12, 25, 44]), whether elements of the topology fit power-law distributions common to large graphs (to inform potential artificial topology generators), and the predictability of individual request workflows (to inform the vast number of tools that work by aggregating trace data [14, 29, 45]). We report on characteristics described in previous studies, such as workflows' sizes and shape, to enable qualitative comparisons.

We perform our study using production datasets[1] describing Meta's microservice topology and request workflows within it.

[1] https://github.com/facebookresearch/distributed_traces

Our datasets for topological analyses span a 22-month period (the entire amount of time historical data about the topology has been maintained). We focus on 1-day of distributed traces [16] (totaling 6.5 million) for our analyses of request workflows, which allows us to focus on predictability of specific request behaviors.

We present our main findings below and conclude this paper with their implications along with a discussion of future research opportunities.

(1) Topological characteristics: The topology is very diverse containing many types of software entities that are deployed as services. The topology is constantly growing, sees daily churn in deployed and deprecated services, and (mostly) does not exhibit power-law relationships.

(2) Workflow characteristics: Traces of request workflows vary in size depending on the high-level functionalities they represent. Similar to previous studies [20, 41], we find that traces are small in size and wide in number of communication calls. Service and endpoint names do not predict number of communication calls or their concurrency. But, they reduce uncertainty in the set of services they will call (callers and callees are specified as services + ingress endpoint). Adding knowledge of the children service set better predicts concurrency.

(3) Obfuscating factors preventing quantitative comparisons between architectures: Scale and complexity analyses are hindered because the term "service" is ill-defined for microservices and previous studies do not report their definitions. Different organizations use different tracing platforms with unspecified assumptions about how workflows are sampled and what sampling policies are used. We find that these factors have non-negligible effects on our results.

2 Toward characterizing Meta's microservices

Figure 1 illustrates Meta's microservice architecture. It is similar to other large-scale microservice architectures [11, 20, 22, 31, 41], consisting of ① (in Figure 1) a topology of interconnected, replicated software services running in dozens of datacenters; ② load balancers for distributing requests amongst service replicas; ③ an observability framework for monitoring the topology and creating traces (graphs) of a sampled set of request workflows; and ④ a globally-federated scheduler for running services on host machines within containers. A basic assumption of Meta's architecture (which may or may not be true for other organizations' architectures) is that *business use case* is a sufficient partitioning by which to define services, scale functionality, and observe behaviors.

The rest of this section motivates the value of studying the topology and request workflows, discusses limitations of previous studies, and fills in important details about Meta's architecture relevant to our analyses. We conclude by discussing the observability framework and the datasets generated from it that we use in our analyses. Given the sparsity of information

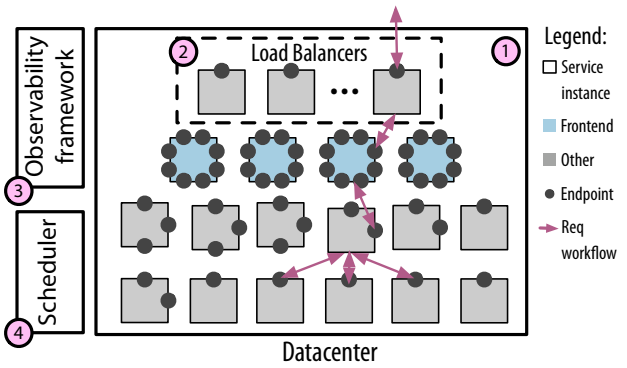


Figure 1: **Meta's microservice architecture and an example request workflow.** The architecture consists of many service instances distributed across many datacenters.

available about Meta's microservice architecture, we err on the side of providing more information than strictly needed.

How do applications use Meta's microservice architecture? Customer applications, such as Instagram or Facebook mobile, issue requests that are load balanced by DNS to specific datacenters and processed by a subset of the architecture's software services. Example requests include those to save photos or record reactions to posts. Applications internal to Meta, such as dashboard or internal tools, use the architecture similarly. But, their requests are load balanced via internal mechanisms, not DNS.

2.1 Topology: services & communication

Why study microservice topologies? We need to understand their complexity, factors that influence their complexity, heterogeneity of constituent services, and the speed at which the topology changes. These characteristics are important to inform tools that visualize the topology, learn models based on the topology, or make assumptions about services' homogeneity [12, 14, 25, 44].

Limitations of existing studies: Only Wen et al. [41] focuses on the microservice topology. The scale they report for number of services is based on a sampled dataset of request workflows, which may not reflect the true scale of their architecture. No existing study defines what constitutes a service or how their definition impacts analyses of the topology (e.g., number of services and communication edges). Existing studies do not report on how the topology evolves or the velocity of change [20, 45].

Meta's microservice topology: The topology is formed by many replicated software services (☐ in Figure 1) deployed across dozens of geographically-distributed datacenters along with their communication to process application requests. (Replicas are typically called *instances*). We note that within the topology, the notion of an application is ill-defined. Individual service instances may process work on behalf of multiple applications. They may also issue requests with batched data from multiple applications to other service instances. The topology evolves *organically* with no central coordination

because development teams responsible for services have complete control over how they are built and maintained.

Services: Services are defined as units of software with well-defined API interfaces, called endpoints (● in Figure 1). Each service satisfies a specific *business use case* (e.g., caching a photo feed). There is significant room for interpretation in defining the scope of a business use case. Additionally, software that pre-dates the microservice architecture may serve multiple business use cases, but be deployed as a single service. Both services and endpoints are named by respective services' developers. (We use *Service ID* and service name interchangeably in latter sections to refer to services' names.)

Services can be stateful or stateless [11]. Stateful services, such as databases, persist state for callers whereas stateless ones, such as search frontends, call other services and integrate their results. A variety of programming languages are used to write services depending on fit for the business use case and societal pressures within the organization.

Load-balancing & Communication: Requests to services are load balanced across their instances. Initially, a datacenter load balancer, itself a service, load balances incoming application requests. Afterward, requests between service instances are load balanced by a service-routing library [30] either linked to applications or to outbound sidecar proxies. (Only some services use sidecars, e.g., when their runtimes do not support linking the routing library directly). The routing library periodically communicates with a global service registry to discover services and routes for their instances. Requests can be load balanced to instances within the same datacenter or to instances in different datacenters. Only the datacenter load balancer is depicted in Figure 1.

Most services at Meta use two-way Thrift RPCs [35] for communication, with payloads serialized in Thrift binary format. Many frontend and some backend services also expose numerous HTTP (REST and GraphQL) endpoints; however, they do not have canonical names that we can use for our analyses. For this reason, we limit the endpoint analysis only to Thrift RPCs reported in the dataset from the routing library.

2.2 Individual request workflows

Why study request workflows: We need to understand the dynamic nature of request workflows. Given a single request execution, what will vary in subsequent executions versus what will remain stable? How much of a statement can we make about other request executions after seeing one or a limited number of samples? Such information is important to inform tools that predict performance, extract critical paths, and present aggregate analyses of request workflows.

Limitations of existing studies: Luo et al. [20] present a way to predict the total number of services that will be called at any hierarchical level of a request workflow. But, they do not discuss whether the number, set, or concurrency level of services called by a specific parent can be predicted. Wen et al. [41] present the amount of time children execute concurrently. Zhang, et al. [45] present distributions of the maximum number of concurrent services observed in workflows. But, neither discuss if information in request workflows can predict concurrency or other workflow characteristics.

Request workflows at Meta: Requests from external applications originate at a datacenter load balancer. This load balancer sends requests to instances of *frontend services*, which are entry points for executing request business logic. There are several frontends at Meta serving different subsets of applications and each has many instances. Frontends may call many services, which in turn may call other services. The resulting hierarchy can be described as forming parent/child relationships. Request workflows for requests originating from internal applications are similar, but originate at the first service that executes business logic on behalf of them.

The set of services involved in a request workflow depends on a number of factors including (but not limited to) the business logic that must be executed on behalf of application requests and whether any requested data is cached. On the other hand, the specific set of instances involved in a request workflow depends on the current load and the load-balancing policy in use.

Concurrency & latent work: Within request workflows, parent services may call all or a subset of children services sequentially or concurrently. The former will be the case if parents are blocking (e.g., single threaded so cannot do work while there is an outstanding call). It may also be the case if there are data dependencies between subsequent calls to children, such as an authentication token that must be returned from one service and passed as input to others. The critical path of concurrently-called children services includes only the slowest one, whereas that of sequentially-called ones includes all of them. Children may perform additional, *latent* work after replying to the parent (e.g., for garbage collection or data replication).

Sample request workflow: The arrows (↔) in Figure 1 show a request traversing a single datacenter. The request first arrives to an instance of the datacenter load balancer, which routes it to an instance of a frontend service, such as `www`. The request then traverses deeper into the topology to backend services.

2.3 Observability framework & datasets

Meta's observability framework includes monitoring mechanisms for recording metrics, logging mechanisms for recording various events, and a distributed-tracing infrastructure, Canopy [16], for recording graphs (called traces) of request workflows. Data generated by the framework is retained for a limited time period to reduce storage volume and due to policy. We describe Canopy in more detail below due to its criticality to observability of microservices. We conclude with a description of the log-based and trace-based datasets we use for our analyses.

Canopy for recording request workflows: Canopy works similarly to most existing distributed-tracing infrastructures [27]. It provides APIs that developers use to define

a request workflow and capture important information about the workflow that should be recorded in traces. The former involves modifying services' code to propagate per-request context—e.g., request IDs and happens-before relationships—within and among the services involved in request execution. The latter involves adding *trace points*, similar to log messages, within source code. During runtime, records of trace points executed by requests are annotated with request context and timestamps. Off of the critical path of request execution, records with identical trace IDs are ordered by happens-before relationships to create traces.

Under the hood, Canopy's implementation is similar to *event-based* tracing infrastructures [10, 26, 29]. However, the way developers instrument services and use the resulting traces is similar to *span-based* tracing infrastructures [4, 32, 34]. *Implementation*: *(1)* Trace points are single events. Higher-level blocks demarcating various intervals (e.g., service executions, queuing time, or function executions) are constructed via annotations added to them. *(2)* context is propagated on both request (forward) and response (reverse) paths, allowing points to be ordered globally within and across services. *Usage*: *(1)* developers (typically) only add blocks denoting service executions; *(2)* happens-before relationships are only established in the forward direction of context propagation, meaning they identify parent/child relationships between blocks and not ordering between siblings; (3) causality between sibling blocks is not explicitly captured via alternate mechanisms. It is impossible to tell whether siblings that execute sequentially as per timestamps in one trace must execute sequentially in other traces.

We describe aspects of Canopy relevant to our workflow analyses. A key observation is that traces created with Canopy may—*by design*—not capture all of a request's workflow.

Effective trace model: Traces are graphs. Nodes are blocks (spans) indicating service execution and hierarchical levels indicate parent/child relationships. Blocks include trace points indicating message send and receives. They may contain additional points indicating other events of interest. Edges between points represent network communication. Latent work started on behalf of a request after the response is returned to the client, such as data replication or asynchronous notifications, may be recorded as additional points on the service block, or as a separate trace with a link back to the originating request's trace (similar to OpenTelemetry's *span links* [5]). Service blocks automatically record *Service ID*s and endpoint names for communication using Thrift RPCs [35]. Developers must manually provide names for services that use custom communication methods. Figure 2 shows an example Canopy trace originating at the *www* service. It has two children services. One of the children (Service B) also has a child (DB).

Streaming model for trace creation with timeout: A stream-processing framework [21] is used to construct traces from trace-point records for subsequent post-processing, such as computing critical path or generating end-to-end latency metrics. The framework accumulates trace events using a

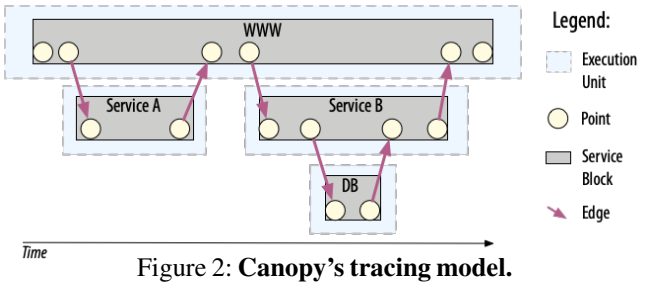


Figure 2: **Canopy's tracing model.**

session window with a fixed gap of inactivity [6]. Traces whose events have a gap in the arrival time larger than the session window would be accumulated in more than one session, but only the first one would be used to trigger post-processing, which may result in processing of partial traces.

Per-service sampling profiles (policies) with rate limiting: Sampling profiles are unique to Canopy. They can be attached to any service and indicate sampling policies to apply based on specific attributes of incoming requests. Traces reflect the union of all sampling profiles that their corresponding requests encounter while executing. This means that a request's trace may start at a service deep in the topology, not recording prior services executed by the request. Trace branches may end prematurely at services whose profiles chose to stop recording the rest of the request's workflow.

A policy specifies: *(1)* a set of conditions for when it is applicable, such as group of endpoints, *(2)* a sampling method, *(3)* a maximum rate of trace data, measured over a sliding time window, beyond which additional traces will not be captured, and *(4)* a verbosity level to decide which instrumentation to execute for requests. Sampling methods may be random head-based sampling [34], in which requests are traced with a random probability, or adaptive sampling [33], in which the sampling probability is periodically changed to achieve a target rate of trace throughput.

Inferred service blocks: These blocks represent services that prematurely ended trace branches, either because of rate limiting or because they lacked tracing instrumentation. Inferred blocks are created during trace construction using information in parent services' message-send points. Inferred blocks may be named or unnamed. The former will be the case when parent points contain the necessary naming information.

Datasets used for this paper: Table 1 shows the datasets we use. For our topological analyses, we use logs describing service activity: history of deployments and deprecation, endpoints exposed by deployed services, and calls made from/to deployed services. For the workflow analyses, we use distributed traces collected by Canopy. The log data describes every deployed service, whereas traces are sampled using methods unique to Canopy, described above.

3 Topological Characteristics

We characterize Meta's current microservice topology as well as how it has evolved. Our analyses of the current topology uses the last (most recent) day of the *Service History* and

Dataset	Description	Format	Retention	Size	Period Used
Service History	Service deployment, lifetimes & interservice communication	*Service ID*s deployed each day	22 months	17 MB	All
Service Endpoints	Endpoints exposed by services	*Service ID* endpoints accessed each day	30 days	18.8 MB	1 day
Traces	Distributed traces	Canopy Trace Objects	30 days	13.1 PB	1 day (4.6 TB)

Table 1: **Datasets used for analyses.**

Endpoints datasets (2022/12/21). Our historical analyses use all 22-months of data available to us (2021/03/01 to 2022/12/21). We also use various dashboards w/statistics about services. The main findings are summarized below.

Finding F1 (All subsections): Meta's microservice topology contains three types of software entities that communicate within and amongst one another: *(1)* Those that represent a single, well-scoped business use case. *(2)* Those that serve many different business cases, but which are deployed as a single service (often from a single binary); *(3)* Those that are ill-suited to the microservice architecture's expectations that business use case is a sufficient partitioning on which to base scheduling, scaling, and routing decisions and to provide observability. These latter entities use *Service ID*s in custom ways, obfuscating their true complexity.

Finding F2 (§3.2): The topology is very complex in its current state, containing over 12 million service instances and over 180,000 communication edges between services. Individual services are mostly simple, exposing just a few endpoints, but some are very complex, exposing 1000s or more endpoints. The overall topology of connected services does not exhibit a power-law relationship typical of many large-scale networks. However, the number of endpoints services expose does show a power-law relationship.

Finding F3 (§3.3): The topology has scaled rapidly, doubling in number of instances over the past 22 months. The rate of increase is driven by an increase in number of services (i.e., new functionality) rather than increased replication of existing ones (i.e., additional instances). The topology sees daily fluctuations due to service creations and deprecations.

3.1 Existence of ill-fitting software entities

We discovered several anomalous patterns in the structure of service IDs within both datasets. For example, we found that on average, 60% of services observed on any single day of the 22-month period have *Service ID*s of the form `inference_platform/model_type_{random_number}`. We found that these services all expose a small number of endpoints with identical names. Meta's engineers informed us that these *Service ID*s are generated by a general-purpose platform for hosting per-tenant machine-learning models (called the Inference Platform). The platform serves a single business use case—i.e., serving ML models—but many per-tenant use cases. Platform engineers chose to deploy each tenant's model under a separate *Service ID* so that each can be deployed and scaled independently per the tenant's requirements by the scheduler.

Following our discovery of the Inference Platform, we investigated the most frequent *Service ID*s and those with the greatest number of service instances. We found two types of software entities that use *Service ID*s in custom ways: *(1)* platforms, such as the Inference Platform, for which multi-tenancy is an additional dimension that must be considered for scheduling, scaling, routing, or observability; *(2)* storage systems, which must take into account data placement in addition to their business use case(s).

We found that some entities, such as the Inference Platform, appear as many services where each service is a combination of the business use case and the additional dimension(s) of partitioning required. Other entities, such as databases and other platforms, appear as a single service and provide their own scheduling and observability mechanisms. Both types of entities' unique use of *Service ID*s masks their true complexity and skews service- and endpoint-based analyses of microservice topologies (ours and likely previous studies [20, 41]).

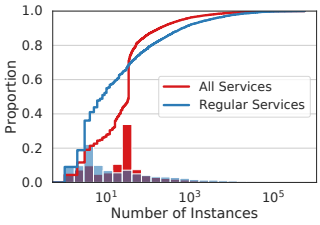
There is no systematic way to identify these ill-fitting software entities at Meta. To illustrate how they may affect *Service ID*-based analyses, we call out contributions by two entities that affect our results significantly. The first is the Inference Platform, which inflates the number of services observed. The second is the ML Scheduler, a scheduling platform for ML training jobs which chooses to appear as a single service and so inflates instance counts. We collectively refer to them and their services as *Ill-fitting services* and all other services as *Regular services*.

3.2 Analysis of the current topology

Scale is measured in millions of instances: On 2022/12/21, the microservice topology contained 18,500 active services and over 12 million service instances. Excluding the ill-fitting services, there are 7,400 services and 11.2 million instances.

The instance count is due to a few highly-replicated services: Figure 3 shows that the ill-fitting services greatly skew instance counts. Notably, the `ML scheduler` is replicated over 270,000 times, 2.2% of all instances. When these services are excluded, the median service's replication factor is only eight and the 99th percentile is 31,306. Frontend service `www` is the most replicated service (557,000 instances, 4.6% of all instances) as it handles most incoming requests.

Services are sparsely interconnected: We construct the service dependency diagram by connecting services that communicate with each other at least once with an edge. (Our dependency diagram is similar to that constructed by OpenTelemetry or Jaeger, except that it is constructed from a portion of the *Service History* dataset that captures commu-

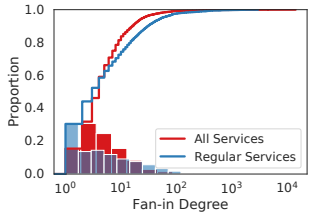


(a) CDF

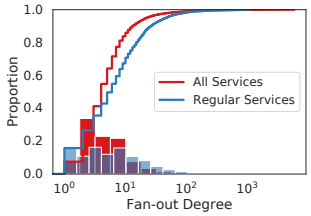
	# Instances	# Instances (Regular)
Min	1	1
Med	34	8
Avg	668	1,504
P99	9,220	31,306
Max	557,156	557,156

(b) Statistics

Figure 3: ***Service ID* replication factors.** The histogram is shown under the CDF. When the curves overlap, the colors are blended together.



(a) Fan-in CDF



(b) Fan-out CDF

Statistic	Fan-in	Fan-out	Fan-in (Reg)	Fan-out (Reg)
Min	1	1	1	1
Median	4	4	3	6
Average	14	12	19	15
P99	86	101	324	158
Max	14,084	5,865	2,968	1,069

(c) Statistics

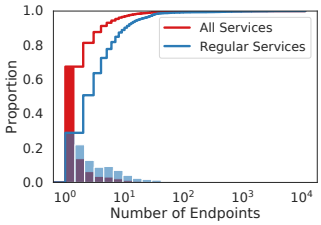
Figure 4: **Service fan-in and fan-out.**

nication between services, not raw traces.) There are 393,622 edges that connect services, which is much smaller than a fully connected topology (18,500^2 or 342 million edges).

Services are called by services more than they call other services: Continuing with the dependency diagram, Figure 4 shows CDFs and statistics about services fan-in (# of services that call them) and fan-out (# of services that they call) degrees. The median fan-in and fan-out are the same, but average and maximum fan-in is larger than fan-outs (14 vs 12 and 14,084 vs 5,865). Excluding the ill-fitting services, by removing all edges connected to ill-fitting services, decreases the median fan-in but increases the median fan-out. Excluding the ill-fitting services also increases the 99.9 percentile and decreases the maximum fan-in and fan-out values.

We investigated the services that have the highest fan-in and fan-out degrees. The former is a vault server storing credentials for use by other services. The latter is a service for querying hosts for arbitrary statistics. Both are used heavily by ill-fitting services, constituting 78% and 91% of the vault service's callers and the services called by the stats service respectively. When ill-fitting services are excluded, two other services rise to the highest fan-in and fan-out degrees respectively: a generic counting service used for various rate limiting mechanisms and a frontend service for internal applications.

Most services are simple, exposing only a few endpoints:



(a) CDF

	# Endpts	# Endpts (Regular)
Min	1	1
Med	1	2
Avg	5	14
P99	26	61
Max	11,359	11,359

(b) Statistics

Figure 5: **Number of endpoints exposed by services.**

Figure 5 shows a CDF and statistics of services' endpoints. Most services do not expose many endpoints (median: 1, P99: 26) and excluding ill-fitting services does not shift the statistics much. The service that exposes 11,359 endpoints is `www`, a frontend service which is used for many business use cases. It is deployed as a single binary from a large, well-engineered codebase that predates the microservice architecture.

Service complexity follows a power-law distribution: Service complexity, measured by number of unique endpoints in a service, follows a power law distribution ($\alpha = 2.23$, $R^2 = 0.99$), indicating that most services are simple with a long tail of more complex services. The power law does not hold for other measures of complexity. Despite there being a long tail of more complex services, the service dependency diagram does not follow a power law distribution ($R^2 = 0.62$). This means the services with more endpoints are not proportionally more connected to the topology than services with fewer endpoints. While there are some highly replicated services, the overall trend of instance counts does not follow a power law distribution either ($R^2 = 0.25$).

Sixteen different languages used to write services: Services can be written in many programming languages. There are currently 16 different programming languages in use at Meta, with the most popular being Hack (a version of PHP), measured by lines of code. Other popular languages include: C++, Python, and Java, with the rest forming a long tail.

3.3 Past growth & dynamism

The number of deployed service instances has almost doubled over the past 22 months: Figure 6 shows the percentage of deployed service instances each day as a percentage of the maximum value observed on 2022/12/21. We show different series for when all services are included, just the ill-fitting services, and only regular services. The slope when all services are considered is $s = 0.052\%$ per day (linear regression $R^2 = 0.95$). The slope when ill-fitting services are excluded is $s = 0.046\%$ per day ($R^2 = 0.95$).

The steady increase in instance counts reflects either an increase in hardware capacity over the time period or an increase in utilization of existing hardware. It cannot be explained by changes in instance sizes as they have remained mostly static.

Instances' rate of increase is due to new business use cases rather than increased scale: Figure 7 shows unique services deployed each day as a fraction of the maximum value

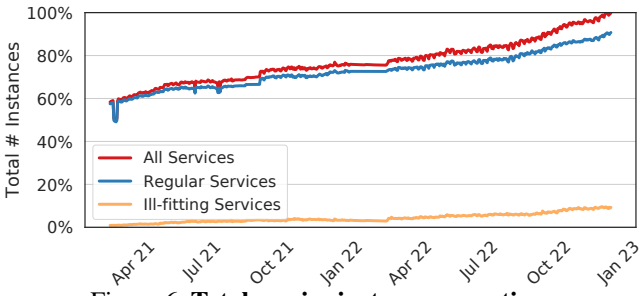


Figure 6: **Total service instances over time.**

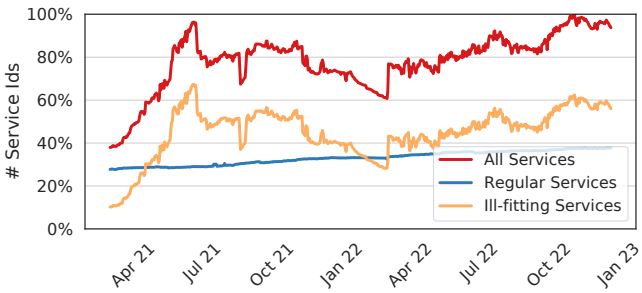


Figure 7: ***Service ID*s over time.**

observed on 2022/11/03. Note that the day with the maximum number of *Service ID*s is different from the day with the most instances. Almost all variability is explained by the ill-fitting services, specifically the Inference Platform, which launches and terminates services as per tenants' demands. The daily increase in services when ill-fitting services are excluded (slope of *Regular Services* series) is $s = 0.043\%$ ($R^2 = 0.98$). It is almost identical to the daily increase in instance counts when ill-fitting services are excluded, which was $s = 0.046\%$ (slope of the *Regular Services* series in Figure 6).

Lots of churn in services, with both long-lived and ephemeral ones: Over the 22-month time period, 180,000 new *Service ID*s were deployed, 89.7% of which were deprecated at some point. Figure 8 shows the number of services created and deprecated each day. Newly-created services are ones whose *Service ID*s were not observed previously during the 22-month period, whereas deprecated ones are services whose *Service ID*s are never observed again. For regular services, creation rates are slightly higher than deprecation rates. As expected, ill-fitting services have high churn.

We also computed the percentage of services observed over the entire period that were deprecated in less than one week (54% of ill-fitting Services, 7.7% of regular services) and the percentage that existed throughout the 22-month period (0% of ill-fitting services, 40% of regular services).

4 Request-workflow characteristics

We now analyze service-level properties of individual request workflows using traces collected by different profiles. We first discuss traces' general characteristics, such as size and width (§4.2). We then analyze whether specific elements of a single trace predict properties of other traces representing the same high-level behavior(s) (§4.3-§4.4). As with any large-scale

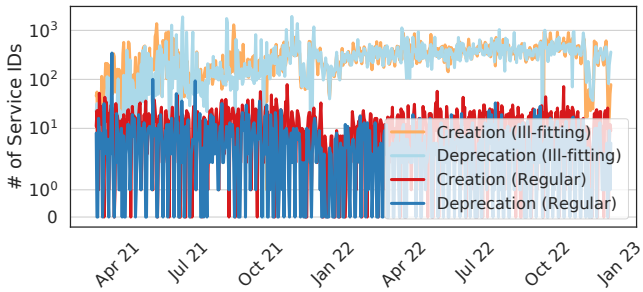


Figure 8: ***Service ID* creation & deprecation.**

tracing infrastructure, traces' visibility into request workflows may be limited due to dropped records, rate limiting, and non-instrumented services. We quantify the extent to which visibility into traces is obscured as a result of these factors in §4.5.

Methodology: For all experiments, we use traces collected on 2022/12/21 from three profiles monitoring important high-level business functionalities. Using a few profiles and a single day avoids factors that would otherwise obscure the interpretability of our results: the effects of analyzing traces using many sampling policies and code updates that change service behaviors. Focusing on important profiles increases the likelihood that traces are representative of their workflows: the services they access are likely to propagate context accurately and use descriptive *Service ID*s and endpoint names. Overall, we analyze 6.5 million traces representing 0.5% of traces collected on 2022/12/21 by all Canopy profiles. Though we do not report them, we observed similar trends to our results on different neighboring days to 2022/12/21 while refining our experiments.

For our predictability experiments, we conduct an ex-post-facto analysis of whether *Ingress ID*s, defined as a combination of *Service ID* and ingress endpoint name, predict properties of their children across many traces. We choose to use *Ingress ID*s because they are readily available in traces, are usually the primary means of understanding trace behaviors, and are location-independent so do not require alignment of traces starting at different (unknown) depths in the topology. We do not consider global characteristics of traces, such as size or width, for prediction experiments as they are not guaranteed to be comparable due to rate limiting or dropped trace records. In our predictability sections, we mean *Ingress ID*s when we refer to parents and children, as in "unique children."

We omit inferred calls (§2.3) from experiments that consider service names since names of inferred services are often unknown. Also, we omit *Ingress ID*s found fewer than 30 times within a profile to allow meaningful statistics to be calculated for the rest of the endpoints.

Our main findings are summarized below; we introduce the profiles afterward. Figure 9 describes the trace properties we analyze and predict.

Finding F4 (§4.2): We measure traces with regard to the number of service blocks they contain (recall from § 2.3 that a service block represents the time interval a service was executed; repeated invocations of the same service appear

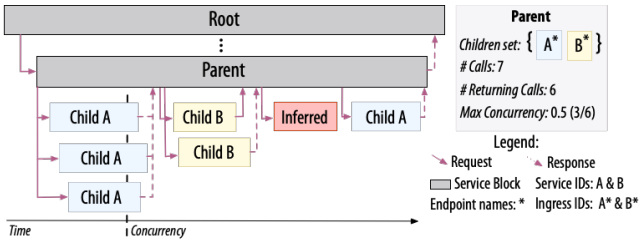


Figure 9: **Trace Characteristics.** Generic example trace showing attributes for a parent *Ingress ID*. Root service is either *www* or *RaaS*. *Inferred* services are represented as blocks of a fixed length since they do not contain notions of time or return edges. They are omitted from concurrency calculations.

as multiple service blocks.) Trace sizes vary depending on workflows' high-level behaviors, but most are small (containing only a few service blocks). Traces are generally wide (services call many other services), and shallow in depth (length of caller/callee branches).

Finding F5 (§4.3-§4.4): Root *Ingress ID*s do not predict trace properties. At the level of parent/child relationships, parents' *Ingress ID*s are predictive of the set of children *Ingress ID*s the parent will call in at least 50% of executions. But, it is not very predictive of parents' total number of RPC calls or concurrency among RPC calls. Adding children sets' *Ingress ID*s to parent *Ingress ID*s more accurately predicts concurrency of RPC calls.

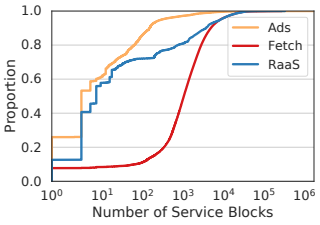
Finding F6 (§4.5): We observe that many call paths in the traces are prematurely terminated due to rate limiting, dropped records, or non-instrumented services. Few of these call paths can be reconstructed (those known to terminate at databases) while the majority are unrecoverable. Deeper call paths are disproportionately terminated.

4.1 Profile details

Ads: This profile represents a traditional CRUD web application focusing on managing customers' advertisements, such as getting all advertisements belonging to a customer or updating ad campaign parameters. The profile captures traces from 56-related endpoints exposed by the `www` frontend service. There are 3.2 million traces over the single-day period. This profile's sampling policy is random at 0.01% capped at 65 traces per second or 160 MB of trace data per minute.

Fetch: This profile represents deferred (asynchronous) work triggered by opening the notifications tab in Meta's client applications. Examples of work include updating the total tab badge count or retrieving the set of notifications shown on the first page of the tab. It captures traces from 91-related endpoints exposed by the `www` frontend service. There are 87,000 traces over the one-day period. This profile uses adaptive sampling with a target rate of 1 trace per second, capped at 20 MB of data per minute.

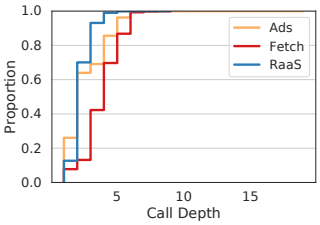
RaaS (Ranking-as-a-Service): This profile represents ranking of items, such as posts in a user's feed. The RaaS sampling policy is applied to the `RaaS` service, a non-frontend service



Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	498	4
Average	83	1,052	628
P99	1,233	8,911	10,667
Max	253,877	81,859	113,790

(a) CDF (b) Statistics

Figure 10: **Trace Size.** Service block counts per trace.



Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	4	2
Average	2.6	3.8	2.3
P99	6	6	5
Max	19	9	9

(a) CDF (b) Statistics

Figure 11: **Call Depth.** Max depth of service blocks per trace.

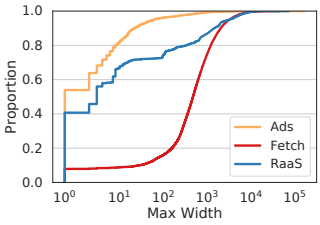
that is called by other services. As a result, traces from this profile always represent only portions of request workflows. Of the workflows we analyze, only those captured by Fetch call `RaaS`. Occasionally, a RaaS trace will be a portion of a Fetch trace, but such occurrences are rare because both Fetch and RaaS profiles use low sampling probabilities that are independent of each other. There are 3.3 million traces over the single-day period, from 4 different endpoints in `RaaS`. This profile uses adaptive sampling with a target rate of 25 traces per second.

4.2 General trace characteristics

There is significant diversity in trace sizes: Figure 10 shows CDFs and statistics of the number of service blocks in our traces. Traces collected by the Fetch profile are significantly larger than Ads and RaaS except at the tail, where Ads traces are largest.

Traces are shallow and wide: Figure 11 shows the maximum call depth in service blocks of our traces starting from trace roots (root is call depth 1). Figure 12 shows maximum trace width, which is the maximum number of calls made by all services at any depth level. (For example, 3 service blocks at one depth making 3 calls each results in a width of 9). We see that across all profiles, traces are much wider than deep: in Fetch profile the median depth is 4 vs. median width of 472, and P99 depth is 6 vs. P99 width of 7,400. We conclude that large traces are a result of the number of calls made by services, not depth of calls. This can be partially explained by the widespread use of data sharding where retrieving a collection of items requires fanning out requests across many storage service instances.

Service reuse within traces is high and occurs at many different call depths: Figure 13 shows CDFs and statistics of the number of services visited within individual traces. Comparing with trace sizes (Figure 10), traces generally contain more service blocks than unique services. At the median, traces visit be-

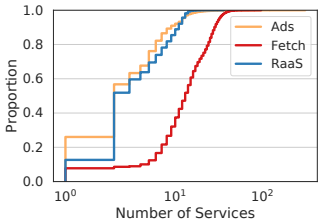


(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	1	472	3
Average	52	916	507
P99	774	7,395	7,885
Max	166,171	46,746	72,196

(b) Statistics

Figure 12: **Max Width.** Maximum number of service calls at a single call depth within a trace.



(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	13	2
Average	4.1	14.6	4.5
P99	18	39	14
Max	307	96	36

(b) Statistics

Figure 13: **Unique Services.** Unique *Service ID*s in a trace.

Type	Ads	Fetch	RaaS
All	421	127	72
Leaf	168 (39.9%)	63 (49.6%)	35 (48.6%)
Single relay	58 (13.8%)	21 (16.5%)	17 (23.6%)
Variable relay	195 (46.3%)	43 (33.9%)	20 (27.8%)

Table 2: **Parent types.** The distribution of parents of each type within each profile.

tween 1x (2/2), 38x (498/13), and 2x (4/2) more service blocks than unique services across Ads, Fetch, and RaaS respectively; at P99 these ratios are 71x, 21x, and 810x respectively.

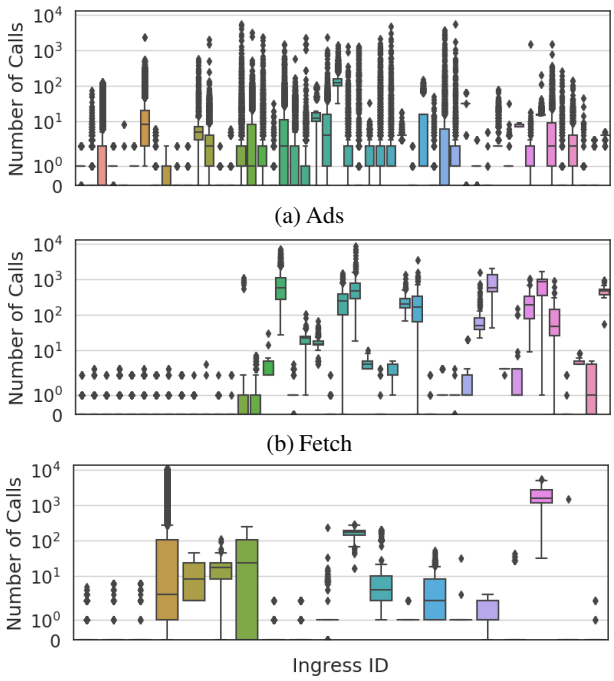
Most services are observed at more than one call depth in our profiles. We measured the number of call depths at which each service was observed. The services in Ads traces have high rates of appearing at multiple depths (median: 6, average: 7.3). Approximately 60% of Fetch and RaaS services are observed at multiple levels (median: 2, average: 2.6 for both profiles).

4.3 Predicting parent/child relationships

Parent *Ingress ID*s strongly predict whether services will have no children or only one child: Table 2 shows that such services, defined as *leaves* and *single relays*, make up from 53 to 73 percent of service executions in our profiles. We find that they are always databases or calls to databases.

***Ingress ID*s do not predict number of downstream calls**: Parents that make one or more downstream calls to children are called *variable relays* in Table 2, making up from 27 to 47 percent of *Ingress ID*s in our profiles. Figure 14 shows that variable relays exhibit a wide distribution in the number of children calls they make. Some *Ingress ID*s exhibit high variance in the number of children they call across different executions whereas others have very little variance (but it is always non-zero).

Variability in number of children calls is due to database calls for Fetch and RaaS: We find that at least 61.1% and



(a) Ads

(b) Fetch

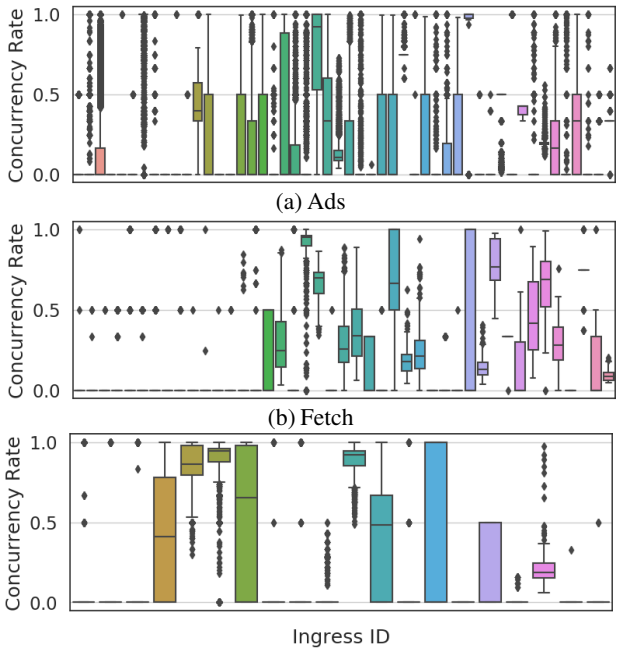
(c) RaaS

Figure 14: **Calls per parent.** Boxplots are shown for every Fetch and RaaS variable relay. Due to limited space, only the 50 variable relays with the greatest number of invocations are shown for Ads. Parent *Ingress ID*s are sorted in descending order by total number of invocations. Boxplot boundaries indicate P25-P75 and the horizontal line within boxes indicate medians. Lower and upper whiskers indicate the smallest/largest data values within 1.5 IQR below/above P25/P75 and dots are outliers.

72.1% of these variable relays' children calls are database accesses in Fetch and RaaS traces respectively. For Ads traces, only 35.7% of children calls are database accesses.

There is a dominant set of unique children per parent: When we ignore number of calls, we find that most single and variable relay parents call only a few *children sets*, where each set is defined as a combination of unique children *Ingress ID*s within a given invocation of the parent *Ingress ID*. For example, one children set may contain `memcache+read` and `database+write`, whereas another may contain `key_service+retrieve` and `database+write`. The average number of children sets called by a relay parent is 28 for Fetch & Ads and 12 for RaaS parents. Most parents have a *dominant children set* that they call in more than 50% of executions. Specifically, 71.9%, 80.2%, and 81.6% of Fetch, Ads, and RaaS relays have dominant children sets.

Non-dominant children sets contain mainly one off children *Ingress ID*s and are not a superset of the parent's dominant children set. On average, only 27% of children *Ingress ID*s called by a parent are in most (>50%) of the parent's children sets.



(a) Ads
(b) Fetch
(c) RaaS

Figure 15: **Parent concurrency.** Concurrency distribution for all invocations of a parent *Ingress ID*. Shows all variable relays for Fetch & RaaS, and the top 50 in Ads by invocation count. Boxplots are interpreted identically to figure 14.

4.4 Predicting children's concurrency

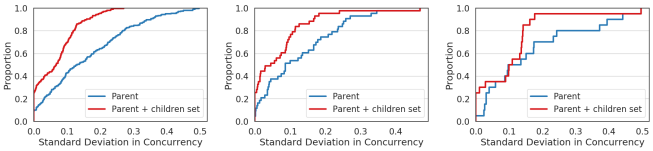
We define *maximum concurrency* as the maximum number of children calls executing concurrently by a parent at any point in time in its execution. More formally, a set of concurrent calls S_t at time t is all children calls with $t_{start} \leq t < t_{end}$, where all timestamps are measured at the parent, and the maximum concurrency C is computed as:

$$C = max(|S_t|), \forall t : t_{start}^{parent} \leq t < t_{end}^{parent} \quad (1)$$

We use a normalized measure of maximum concurrency, the *concurrency rate*, calculated as $C/num\_children$, to allow comparisons across different executions of the same parent (different numbers of children may be called in each execution) and to allow comparisons between different parents. *num_children* refers to children that have return edges and well-defined durations. We only consider variable relays since concurrency is ill-defined for leaves and single relays.

Parent *Ingress ID* does not predict whether children will execute concurrently or sequentially: Figure 15 shows boxplots of concurrency rates across executions for each parent *Ingress ID* observed in our traces. We see that there is a mix of high and low variation in concurrency rate across *Ingress ID*s.

The combination of parent *Ingress ID* and children set more accurately predicts concurrency rate: Figure 16 shows a CDF of the standard deviation in concurrency rate across all executions of parent *Ingress ID*s. To understand if children set adds predictability value, we calculate the standard deviation for each parent's children set and average them to obtain a



(a) Ads (b) Fetch (c) RaaS

Figure 16: **Standard deviation in concurrency rate.** *Parent* shows a CDF the standard deviation in concurrency rates for all executions of a parent. *Per-parent avg. children set* shows the average standard deviation per children set for each parent.

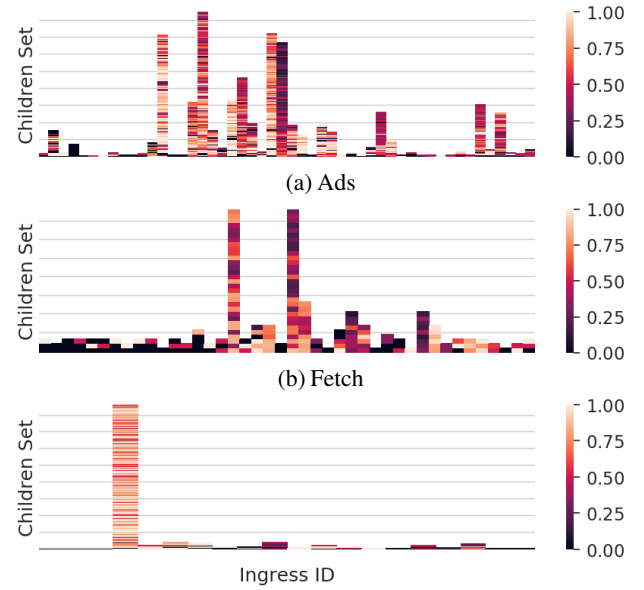
per-parent average. We plot this CDF of per-parent average. Intuitively, if children sets provide value, the per-parent average should decrease whereas if they do not, the data points will be randomly distributed and standard deviation will not decrease. Overall, we find that including children sets shifts the distributions to the left. The shift is most pronounced at the median for Ads and Fetch: 0.13, 0.09 vs. 0.04 and 0.02. Adding children set does not provide value in the tail for Fetch and RaaS.

We speculate the reduction in standard deviation is because children belonging to the same children set likely have well-defined control or data dependencies between each other. Reduction in variation due to control dependencies may be a result of custom threading models for different code logic blocks in parents (each responsible for a different behavior and thus children set). For data dependencies, consider the following examples. Children sets containing different cache services may have no dependencies and thus may be able to execute concurrently. In contrast, children sets comprised of a key server and a database service may have to execute sequentially: credentials may be required to access the database.

***Ingress ID* + children set calls display a range of dependency relationships**: We now quantify the strength of dependencies within *Ingress ID* + children set's calls. We use the maximum concurrency rate observed across *Ingress ID* + children set executions as a indicator of dependency. A maximum concurrency rate of 1 implies that there are no dependencies among children calls. A maximum observed concurrency rate of 0 builds confidence that the children calls are dependent and must execute sequentially. Figure 17 shows the results. Overall, we find that most *Ingress ID* + children set executions display weak dependencies (some concurrency) and there are a few strongly dependent (sequential) children sets.

4.5 Quantifying traces' observability loss

Most prematurely terminated call paths are unrecoverable: We plot the percent of branches that terminate prematurely (at an *inferred* service) at each call depth in our traces (Figure 18). Some branches terminate prematurely due to internal rate-limiting at databases, which are usually leaves in the traces. The shaded area in Figure 18 is the portion of inferred services that represent known databases. The distance between the curves are unknown inferred services, which make up the majority of inferred services. Using trace data alone, we cannot know what the unknown inferred services are



(a) Ads

(b) Fetch

(c) RaaS

Figure 17: **Parent *Ingress ID* + children set max concurrency rate.**

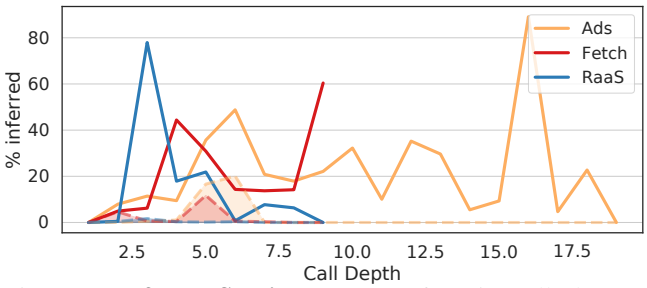


Figure 18: **Inferred Services.** Percent of service calls that are inferred at each call depth. The shaded region is the percent of inferred services that are known to be database calls.

(some may still be other databases we were not able to identify reliably) or the shape of the workflow from that point on.

Non-uniform probability a branch is terminated: Deep branches are disproportionately prematurely terminated. For RaaS traces, 80% of call paths that reach depth 3 are terminated with inferred nodes, none of which were identified as databases. However, as the average trace depth for RaaS is only 2.3 (Figure 11), the majority of RaaS traces are not affected by premature branch terminations. Similarly, Fetch and Ads traces are shallow and prematurely terminated branches mainly occur beyond the average trace depth.

5 Implications and opportunities

Implications for microservice testbeds: Existing testbeds [1, 13, 46] represent only single applications, whereas microservices within Meta serve many applications (§2.1). Previous studies state that existing open-source testbeds' topologies are lacking in scale compared to industrial microservices [20, 41]. Our results confirm these results (Finding F2) and add the following dimensions to consider in future testbeds: heterogeneity of services, churn, and growth. Specifically, we find that Meta's microservice architecture contains a mix of software entities that are deployed as services: complex ones that expose many endpoints and are likely more monolithic in nature, simple ones that expose just a few, and ill-fitting ones that require support beyond which the microservice architecture provides by default (Finding F1). We find that services are deployed and deprecated (at least) daily and that the shape of the communication topology is constantly growing and changing (Finding F3).

Luo et al. [20] state that request workflows within existing testbeds are too static. Many service-level workflow properties can be predicted from root endpoint alone. Our analyses show that future testbeds should include concurrency, number of children, and set of children that are executed as dimensions of variability in request workflows representing the same or similar high-level behaviors (Finding F5).

Implications for microservice tooling: Tools that use models of microservice topologies [12, 25, 44] should assume that its constituent services are always changing and that the topology itself is highly-dynamic (Finding F3). Periodic retraining may be necessary; mechanisms are needed to identify when predictions diverge from the ground truth due to stale topological information.

Tools that aggregate request-workflow traces for performance predictions, diagnosis, or capacity planning [7, 14, 28, 29, 45] must assume that there is significant diversity in workflows originating from the same root endpoints or groups of related root endpoints (Finding F5). Our studies show that many workflow properties can be predicted when they are broken down into fundamental building blocks (parent/child relationships) (Finding F5), perhaps a promising starting point for aggregation-based tools. However, capturing total orderings for entire traces [14] or even individual services may not scale due to parent *Ingress ID*s initiating large number of RPCs with high concurrency (§4.4).

Need for artificial microservice topology & workflow generators: Such generators are a necessity given the infeasibility of creating microservice deployments outside of industrial settings. The sole existing workflow generator [20] may be too specific to a single organizations' microservice design (that number of children depends on depth in trace) and generates stochastic workflows that do not represent any single request. Research is needed to identify: *(1)* which dimensions of microservice architectures are best explored in testbeds versus artificial topology or workflow generators; *(2)* how to ensure these dimensions are representative of a variety of large-scale organizations' characteristics. Our analysis shows that assuming topologies follow power-law relationships is insufficient for modeling microservice topologies (Finding F2).

How to better incorporate ill-fitting software entities into microservice architecture? Ill-fitting entities constitute a significant portion of Meta's microservice topology. Key questions include: *(1)* Should infrastructure platforms provide richer interfaces to allow scheduling, scaling, and observability based on additional dimensions rather than only one? *(2)* In

cases where ill-fitting-entities use custom techniques, what mechanisms are necessary to allow mapping them to standard service-level operations?

Naming & predicting missing elements of workflows: Our predictability results (Finding F5) indicate that well-defined service and endpoint names are important for predicting local workflow properties. Almost all tools that use distributed traces [7, 12, 14, 25, 28, 29, 42, 44, 45] assume descriptive names. But, naming quality can vary considerably, especially for services that satisfy many business use cases and for microservice architectures in which all instrumentation is done within proxies surrounding services [3]. Research into naming schemas that allow different parts of service behaviors to be differentiated based on parts of the name (or attached attributes) is needed. Research is also needed into how to automatically identify meaningful names and/or attributes that differentiate important within-service behaviors, and whether missing observability data (Finding F6) can be predicted based on other data already available.

Need for standardized methods to contrast different organizations' microservice architectures: Our original goal for this research was to compare characteristics of Meta's microservice architecture with previous studies of industrial microservice architectures. At 30,000 ft, we find that organizations' architectures have similar architectural diagrams (Figure 1) and use custom versions of the same architectural components or open-source versions [4, 19, 35]. Furthermore, similar to Meta, the traces used in Luo et al. [20] and Wen et al. [41] tend to be small. Large traces are wider than deep, indicating common use of data sharding. We also find some differences. Traces used in Zhang et al. [45] seem to be much deeper than those used in our analyses, perhaps due to their domain-oriented microservice strategy [15].

Unfortunately, we found more detailed quantitative comparisons to be impossible due to divergent (or ill-specified) definitions in previous studies and because different studies use custom measurement techniques specific to their observability frameworks. With regard to comparing scale and complexity, previous studies do not define the term *service*, describe individual service's complexity, or describe number of communication edges between services, or service instances. For request-workflow-based analyses, these studies do not identify tracing sampling rates and mechanisms, whether traces capture all of request workflows or only parts or whether dropped records or rate limiting impact their analyses. Similar to rich research into Internet measurement [2], we need to develop rich, well-accepted methodologies for collecting data about microservice architectures to understand and systematize similarities and differences across them.

6 Summary

The characteristics of large-scale microservice architectures are largely invisible outside of industrial organizations. We presented an analysis of Meta's microservice architecture to inform more robust assumptions for future microservices research and development.

7 Acknowledgments

We thank our shepherd, Jan Rellermeyer, the anonymous USENIX ATC conference reviewers, Lenni Kuff, Alex Knott, and members of Meta's observability team for their insightful feedback and support. We thank Theo Benson for suggesting examplars on which we could base this paper's structure.

References

[1] BookInfo application. URL: https://istio.io/latest/docs/examples/bookinfo/.

[2] IMC: Internet Measurement Conference. URL: https://dl.acm.org/conference/imc.

[3] ISTIO: Simplify observability, traffic management, security, and policy with the leading service mesh. URL: https://istio.io.

[4] OpenTelemetry. URL: https://opentelemetry.io/.

[5] Tracing in OpenTelemetry. URL: https://opentelemetry.io/docs/concepts/signals/traces/.

[6] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming systems: the what, where, when, and how of large-scale data processing*. O'Reilly Media, Inc., 2018.

[7] Vaastav Anand, Matheus Stolet, Thomas Davidson, Ivan Beschastnikh, Tamara Munzner, and Jonathan Mace. Aggregate-driven trace visualizations for performance debugging. *CoRR*, abs/2010.13681, 2020. URL: https://arxiv.org/abs/2010.13681, arXiv:2010.13681.

[8] Adrian Cockroft. The evolution of microservices, 2016. URL: https://learning.acm.org/techtalks/microservices.

[9] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Migrating towards microservice architectures: an industrial survey. In *ICSA'18: Proceedings of the International Conference on Software Architecture*. IEEE, 2018. doi:10.1109/ICSA.2018.00012.

[10] Rodrigo Fonseca, Michael J. Freedman, and George Porter. Experiences with tracing causality in networked services. In *INM/WREN'10: Proceedings of the 1st Internet Network Management Workshop/Workshop on Research on Enterprise Monitoring*, 2010.

[11] Martin Fowler. Microservices: a definition of this new architectural term. URL: https://martinfowler.com/articles/microservices.html.

[12] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ML-driven performance debugging in microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[13] 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 *ASPLOS'19: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[14] Lexiang Huang and Timothy Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[15] Introducing Domain-Oriented Microservice Architecture. URL: https://www.uber.com/blog/microservice-architecture/.

[16] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *SOSP'17: Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[17] Justas Kazanavičius and Dalius Mažeika. Migrating legacy software to microservices architecture. In *eStream'19: Proceedings of the Open Conference of Electrical, Electronic and Information Sciences*. IEEE, 2019. doi:10.1109/eStream.2019.8732170.

[18] Tom Killalea. The hidden dividends of microservices. *Communications of the ACM*, 59(8):42–45, 2016.

[19] Kubernetes. URL: https://kubernetes.io.

[20] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[21] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, et al. Turbine: Facebook's service management platform for stream processing. In *ICDE'20: Proceedings of the 36th International Conference on Data Engineering*, pages 1591–1602. IEEE, 2020.

[22] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2nd edition, 2021.

[23] HL Phalachandra, Pranav Bhatt, Ishitha Agarwal, and Rishith Bhowmick. Improving task scheduling in microservice environments by considering intra-job dependencies. In *ICICCS'22: Proceedings of the 6th International Conference on Intelligent Computing and Control Systems*, pages 568–573. IEEE, 2022.

[24] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *SCCC'19: Proceedings of the International Conference of the Chilean Computer Science Society*. IEEE, 2019. doi:10.1109/SCCC49216.2019.8966423.

[25] 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 *OSDI'20: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 805–825, 2020.

[26] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul Shah, and Amin Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006.

[27] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. Principled workflow-centric tracing of distributed systems. In *SoCC'16: Proceedings of the Seventh Symposium on Cloud Computing*, 2016.

[28] Raja R. Sambasivan, Ilari Shafer, Michelle L Mazurek, and Gregory R. Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2466–2475, 2013.

[29] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[30] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: a Scalable and Minimal Cost Service Mesh. In *OSDI'23: Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[31] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R Sambasivan. [SoK] Identifying mismatches between microservice testbeds and industrial perceptions of microservices. *Journal of Systems Research*, 2(1), 2022. doi:http://dx.doi.org/10.5070/SR32157839.

[32] Yuri Shkuro. Jaeger: Evolving distributed tracing at Uber Engineering. URL: https://www.uber.com/blog/distributed-tracing/.

[33] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing, 2019.

[34] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, April 2010.

[35] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 2007. URL: http://thrift.apache.org/static/files/thrift-20070401.pdf.

[36] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[37] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[38] Mert Toslali, Srinivasan Parthasarathy, Fabio Oliveira, Hai Huang, and Ayse K Coskun. Iter8: Online experimentation in the cloud. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[39] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):1–44, 2021.

[40] Hao Wei, Joaquin Salvachua Rodriguez, and Octavio Nieto-Taladriz Garcia. Deployment management and topology discovery of microservice applications in the multicloud environment. *Journal of Grid Computing*, 19(1):1, 2021. doi:10.1007/s10723-021-09539-1.

[41] Yingying Wen, Guanjie Cheng, Shuiguang Deng, and Jianwei Yin. Characterizing and synthesizing the workflow structure of microservices in ByteDance cloud. *Journal of Software: Evolution and Process*, 34(8):1–18, 2022.

[42] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *ICSE'22: Proceedings of the 44th International Conference on Software Engineering*, 2022. doi:https://doi.org/10.1145/3510003.3510180.

[43] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 3Milebeach: A tracer with teeth. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[44] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[45] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *ATC'22: Proceedings of the 2022 USENIX Annual Technical Conference*, 2022.

[46] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Poster: Benchmarking microservice systems for software engineering research. In *ICSE'18 Companion: Proceedings of the 40th Companion to the International Conference on Software Engineering*, 2018.

Tectonic-Shift: A Composite Storage Fabric for Large-Scale ML Training

Mark Zhao[1,2], Satadru Pan[2], Niket Agarwal[2], Zhaoduo Wen[2], David Xu[2], Anand Natarajan[2], Pavan Kumar[2], Shiva Shankar P[2], Ritesh Tijoriwala[2], Karan Asher[2], Hao Wu[2], Aarti Basant[2], Daniel Ford[2], Delia David[2], Nezih Yigitbasi[2], Pratap Singh[2], Carole-Jean Wu[2], Christos Kozyrakis[1]
[1]*Stanford University,* [2]*Meta*

Abstract

Tectonic-Shift is the storage fabric for Meta's production machine learning (ML) training infrastructure. Industrial storage fabrics for ML need to meet both the intensive IO and high-capacity storage demands of training jobs. Our prior storage fabric, *Tectonic*, used hard disk drives (HDDs) to store training data. However, HDDs provide poor IO-per-watt performance. This inefficiency hindered the scalability of our storage fabric, and thus limited our ability to keep pace with rapidly growing training IO demands.

This paper describes our journey to build and deploy *Tectonic-Shift*, a composite storage fabric that efficiently serves the needs of our training infrastructure. We begin with a deep workload characterization that guided an extensive hardware and software design space exploration. We then present the principled design of *Tectonic-Shift*, which maximizes storage power efficiency by combining *Shift*, a flash storage tier, with *Tectonic*. *Shift* improves efficiency by absorbing reads using IO-efficient flash, reducing required HDD capacity. *Shift* maximizes IO absorption via novel application-aware cache policies that infer future access patterns from training dataset specifications. *Shift* absorbs $1.51-3.28\times$ more IO than an LRU flash cache and reduces power demand in a petabyte-scale production *Tectonic-Shift* cluster by 29%.

1 Introduction

The success of industrial machine learning (ML) training is enabled by highly efficient and scalable infrastructures that store and feed massive amounts of training data to datacenter-scale training clusters [27, 31, 44, 46, 68]. At Meta, we deploy training clusters, each with of thousands of GPUs, across many datacenters in order to meet our ML training demands [42]. Each cluster requires a storage fabric capable of storing exabytes of data and serving reads at tens of terabytes per second.

Our prior storage fabric was *Tectonic*, Meta's exabyte-scale distributed file system [50]. Each *Tectonic* instance is backed by a cluster of disaggregated hard disk (HDD) storage nodes. To feed trainers, we needed to provision each *Tectonic* cluster with HDDs to provide *both* sufficient storage capacity for training datasets and enough IO capacity to meet the read bandwidth demands of all trainers in the datacenter. The significant and increasing IO requirements of training accelerators resulted in a large imbalance between IO and storage demands compared to what is afforded by modern HDDs. We needed to provision an order of magnitude more storage capacity to meet trainers' IO demands than to store datasets. This storage inefficiency expended a large portion of each datacenter's power budget — *modern ML storage fabrics often require more power than trainers themselves* [68] — which constrained the scalability of our training infrastructure.

This paper chronicles our journey to improve the power efficiency of our production storage fabric for IO-bound ML training workloads. We begin with a hardware design space exploration and show that traditional homogeneous storage fabrics (HDD or otherwise) cannot meet the imbalanced storage and IO demands of ML training without resource over-provisioning. An ideal storage solution should combine multiple storage media in a *composite storage fabric* to balance storage and IO capacity. It can efficiently meet IO demands by serving most IOPS from IO-efficient (high bytes/s per watt) devices, e.g., flash, while relying on storage-efficient (high bytes per watt) devices, e.g., HDDs, to meet storage demands.

However, simply deploying a composite storage fabric does not beget high efficiency. It must hold the *right data* in IO-efficient devices at the *right time* — exploiting data locality via caching. We present a software design space exploration, guided by a deep characterization of our production ML training workloads, showing that current cache systems do not capture the data reuse characteristics of these workloads. General-purpose software flash and DRAM caches are designed for web-based workloads with trillions of small requests such as content delivery networks, social graphs, key-value stores, and databases [1, 6, 7, 9, 13, 38, 43, 45, 55, 56, 66]. Meanwhile, ML training jobs issue a small number of massive scans over petabytes of data, resulting in scan and churn patterns that easily thrash an LRU cache [52]. Alternatively, current ML-

specific storage systems [16, 23, 30, 32, 41, 61, 71] are ineffective because existing solutions have been designed for small-scale deployments with highly-synchronized training jobs reading multiple epochs of the same static data. Meanwhile, large-scale production training environments consist of highly asynchronous, single-epoch training jobs reading varying subsets of continuously-updated datasets.

While cache systems leveraging composite storage have been widely studied and deployed, our hardware and software design space exploration elucidates the need for a unique combination of techniques tailored to our ML workloads. To this end, we built *Tectonic-Shift*, a composite storage fabric that improves storage efficiency by balancing storage and IO capacity across HDDs and flash. We present *Tectonic-Shift* and several guiding design principles that make it deployable and effective across our datacenters: a) *Transparent. Tectonic-Shift* presents the same APIs as the *Tectonic* File System, requiring no user knowledge or application changes. *Tectonic-Shift* combines *Shift*, a flash storage tier that aims to maximize IO absorption, with each HDD *Tectonic* cluster. b) *Simple. Shift* is built on top of CacheLib [6], and deploying *Shift* requires no changes to other storage services such as *Tectonic*'s Metadata Layer. c) *Scalable. Shift* is fully decentralized, consisting only of disaggregated flash storage nodes, each using local dynamic cache policies that adjust to observed load. d) *Intelligent.* While *Tectonic-Shift* is transparent to users, it understands application information from training job specifications, such as the list of table partitions that comprise the job's dataset. We present novel cache mechanisms that leverage this information to improve the performance of *Shift* by inferring training jobs' future data access patterns.

We demonstrate how these principles allow *Shift* to absorb $1.51-3.28\times$ IO than an LRU-based flash cache on a mix of representative training workloads, all while managing flash endurance limits. Furthermore, we present results on our petabyte-scale production tiers, serving real ML training jobs, showing how *Tectonic-Shift* can save 29% of power relative to using HDDs alone for training data. We close with a discussion of lessons learned in deploying *Tectonic-Shift* and several promising areas of future exploration. In summary, we make the following contributions.

- We provide an in-depth hardware and software design space exploration of storage systems for ML training jobs, guided by a characterization of our production workloads.
- We present the principled design of *Tectonic-Shift*, which combines *Shift*, a flash storage tier, with each *Tectonic* cluster to improve the overall efficiency of Meta's storage fabric.
- We describe novel cache policies employed by *Shift* that predict and optimize for future data access patterns derived from training job specifications.
- We show detailed production evaluation results. *Shift* absorbs $1.51-3.28\times$ more IO than LRU. *Tectonic-Shift* improves the power efficiency of our storage fabric by 29%.

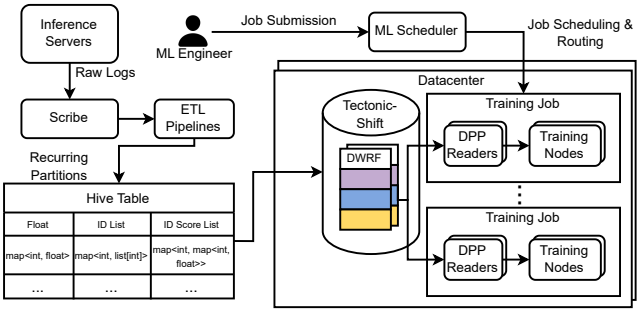


Figure 1: Overview of Meta's data storage and ingestion (DSI) pipeline. *Tectonic-Shift* is the durable storage fabric for training data in each datacenter.

2 ML Data Storage and Ingestion Background

Deep learning recommendation model (DLRM) training dominates our ML infrastructure demands [2], requiring significant data storage and ingestion (DSI) capacities to manage structured datasets [68]. Thus, we primarily focus on DLRM workloads in this paper, and we discuss extending *Tectonic-Shift* to other ML domains and non-ML workloads in Section 7. Figure 1 shows how the DSI pipeline continuously generates, stores, and ingests DLRM training data.

Data Generation. Fresh data is needed to ensure model accuracy [18]. We continuously generate training samples from inference requests served by our production fleet. When a given host serves an inference request, it logs a snapshot of the relevant *features* of the requester (e.g., a user's set of liked pages) and the outcome of the *event* corresponding to the inference request (e.g., if a user likes the recommendation). These logs are continuously published to Scribe [26], Meta's global distributed messaging system. A training data pipeline, corresponding to a set of extract-transform-load (ETL) jobs (e.g., Spark [67]), consume these logs by joining and labeling them to form structured training samples.

Dataset Storage. Each pipeline's training samples are stored in a corresponding Hive [57] table. Tables are constantly updated with new time-based partitions of fresh data, generated by each pipeline with a regular cadence (e.g., hourly). Old partitions regularly expire and are deleted. Each table is replicated to all datacenters with training clusters, and each partition is stored as columnar DWRF [21] files (similar in format to ORC [14]) in a new directory in each respective datacenter's *Tectonic* File System. Training jobs read from their local *Tectonic* instance.

We adopt a common schema across our training tables to ensure interoperability across models [68]. Specifically, all features are stored in a small number of map columns and comprise the majority of each row ($>99\%$ of bytes). Each column maps multiple integer feature IDs to the row's corresponding value for that feature (e.g., a float for a dense feature column or lists/maps for a categorical feature column).

Data Ingestion. Training jobs are submitted to a global queue

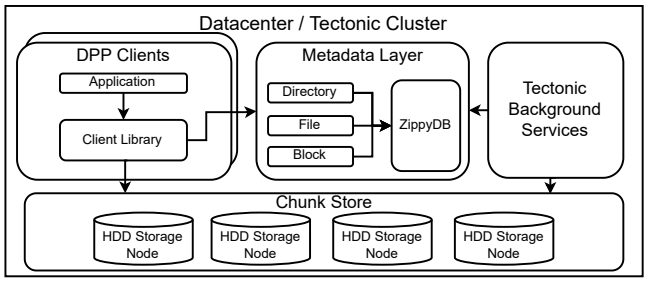


Figure 2: Overview of the *Tectonic* File System. Directory, file, and block metadata operations are served by a Metadata Layer. Clients directly read chunks from HDD-based Chunk Store nodes.

by ML engineers and are scheduled and routed to a specific datacenter when capacity allows. When each job is scheduled, it allocates a set of training nodes (Trainers) [42] and a set of Data PreProcessing (DPP) Readers [68] from the datacenter's training cluster. Trainers are equipped with GPUs and perform the actual training, continuously ingesting tensors from Readers. Readers are general-purpose CPU nodes that continuously read raw bytes from the *Tectonic* File System, reconstruct minibatches of samples from the bytes, and preprocess each minibatch into tensors.

Specifically, Readers read data based on a training job's *dataset*, specified by an ML engineer. The dataset contains a *list of table partitions* and a *list of feature IDs*. Throughout the lifetime of the training job, Readers will continuously ingest (disjoint) minibatches of samples, filtering out unused features in each sample, until the Readers have collectively read all samples from the specified partitions. To read the appropriate bytes from *Tectonic*, Readers map each partition to a set of *Tectonic* files by querying the Hive Metastore [57,58]. Readers then scan each file and progressively read samples by issuing *Tectonic* reads. Readers push filtering to storage, referencing metadata within footers of the file to selectively read bytes corresponding to features specified by the dataset [68].

***Tectonic* File System.** Figure 2 shows the architecture of the *Tectonic* cluster (sans *Shift*) backing each *Tectonic* File System instance. Files are divided into *blocks* (typically 72 MiB) representing a logical array of bytes. *Tectonic* further divides blocks into smaller *chunks* (typically 8 MiB) and durably encodes each via replication or Reed-Solomon (RS) encoding [51]. Chunks are distributed across the cluster's Chunk Store, backed by a number of HDD storage nodes.

Readers directly read data from storage nodes using the *Tectonic* Client Library. The Client Library exposes a file `pread` interface to clients. For each read, the Client Library issues requests to specific chunks on storage nodes and performs reconstructions if necessary. The Client Library obtains chunk mappings and any directory and file metadata (e.g., directory `ls`) via queries to a hash-sharded Metadata Layer built on ZippyDB [37]. DPP Readers optimize for HDD seeks and coalesce reads into large $O(1MB)$-sized IOs [68].

Table 1: Storage power requirements for an HDD, flash, and ideal composite cluster, assuming 100 PB and 10 TB/s storage and IO demand. We show required power to meet storage-only, bandwidth-only, and both requirements, normalized to HDD storage-only.

	Storage Req.	IO Req.	Storage & IO Req.
HDD Cluster	1.00	9.92	9.92
Flash Cluster	6.53	1.88	6.53
HDD + Flash	1.00	1.88	2.69

3 Production ML Storage Design Space

This section explores why we could not efficiently scale *Tectonic* to meet the IO bandwidth that our training clusters increasingly demand. We present various hardware and software design space explorations that led us to *Tectonic-Shift*, guided by a characterization of our production ML training jobs.

3.1 Hardware Design Space

We first evaluated different storage hardware options, summarized in Table 1. Specifically, we used our HDD [3] and flash [8] server specifications to calculate the power (watts) required by the number of HDD, flash, or HDD + flash servers (rows) to supply 100 PB of storage, 10 TB/s of read bandwidth, or both (columns). These demands are representative of our workloads [42,68], and we must provide both sufficient storage *and* IO capacity. The HDD + flash analysis used only HDDs to supply 100 PB of storage and only flash to supply 10 TB/s of IO. In the storage and IO case, we used HDDs to supply storage capacity and flash to supply IO capacity, discounting the IO capacity supplied by the HDDs. We focused on power because it is the primary budget and optimization metric for services across our fleet [68]. We normalized results to the HDD, storage-only case.

Option 1: HDD-Only (Status Quo). Our first option to meet IO demand was to continue provisioning more HDD storage nodes into each *Tectonic* cluster. This would linearly scale IO capacity as chunks are distributed across HDDs evenly. Unfortunately, this option would require us to provision 9.92× more storage capacity than necessary — $1.6EB$ of disks assuming $RS(9,6)$! Furthermore, since our IO demand is growing 2× as fast as storage demand [68], this option is unsustainable.

Option 2: Flash-Only. We also considered using a flash storage tier [28] for our training datasets. Flash trades off storage-efficiency for IO-efficiency. Compared to HDDs, A flash cluster would need 5.28× less power to meet IO demand, but 6.53× more power to meet storage demand. Meeting both demands is more efficient using flash, but there is a significant over-provisioning of IO capacity, making this sub-optimal.

Option 3: Composite Storage. Relying on a single storage hardware inherently precludes us from balancing storage and IO capacity. An ideal cluster would use *both* a storage-efficient device and IO-efficient device together, provisioning enough of each to meet their respective demands. HDDs are

Figure 3: IO bandwidth demand across 85 tables over the course of one day. Training tables exhibit a power law in popularity.

an ideal storage-efficient device due to their density. We considered DRAM and flash for our IO-efficient device.

Option 3.1: HDD + DRAM. We decided against using only DRAM, as a DRAM storage node would be bound by the NIC throughput (e.g., 100 Gbps) as opposed to DRAM throughput. Modern SSDs can provide $O(1GB/s)$ of read bandwidth at $O(1W)$ of power [11], allowing a flash storage node to provide the same IO capacity in roughly the same power footprint as a DRAM storage node. Meanwhile, flash storage nodes have significantly higher storage capacities ($O(10TB)$), greatly improving cache performance.

Option 3.2: HDD + Flash. We opted to use flash as our underlying IO-efficient device. Table 1 shows how an ideal composite cluster would allow us to provision only 1.69× flash-based power to meet IO demand plus 1.00× HDD-based power to meet storage demand, reducing the power footprint of our storage tier by 3.69× compared to Option 1.

However, Option 3.2 assumes that flash servers are able to meet the bulk of IO demand by holding a popular subset of bytes. Our next challenge was designing a system that could intelligently manage the contents of each flash server to maximize its IO absorption. Our first option was to create both a flash and HDD Chunk Store within the same durable storage fabric. The *Tectonic* Metadata Layer would move blocks between flash and HDD based on read demand. This option has several drawbacks. It a) requires extra RS encoding overheads on flash, b) adds metadata overheads due to block location updates, and c) requires significant changes to the Metadata Layer to support sub-block granularities (due to byte-range popularities, to be discussed in Section 3.2.1). For these reasons, we decided to use flash as the foundation for a metadata-less, non-durable cache. We present the design space exploration of this software cache next.

3.2 Software Design Space

3.2.1 Production ML Workload Characterization

We begin by characterizing our production ML training jobs, which present a uniquely challenging cache workload.

Row-wise Reuse. Training samples (rows) exhibit a skewed popularity across training jobs. Figure 3 shows the IO demand targeting 85 tables over the course of one day. We run many ML model types in production, and ML engineers continuously train and experiment on each model type with varying

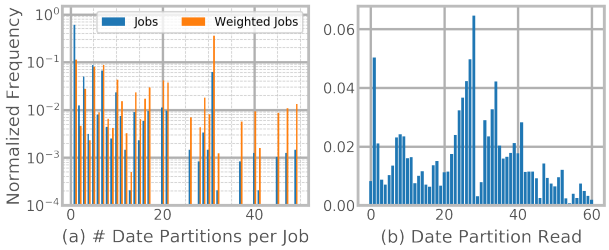


Figure 4: (a) Normalized histogram showing # of date partitions read by each training job, with orange bars weighing jobs by the number of partitions read. (b) IO demand across date partitions over a one-day period, with day 0 being the most recent partition.

popularities. Since each model type typically uses a distinct table, this variation manifests in tables' IO demands. There is a distinct power-law in table popularity with a long tail.

Furthermore, training jobs typically read a subset of table partitions, as models can typically reach convergence before all rows are exhausted. For a similar reason, each training job only reads its specified rows once (i.e., one epoch). Figure 4(a) shows a distribution for a popular table[1] of the number of date partitions[2] read by training jobs in one day. Most jobs read only a few partitions. However, when we consider IO demand by weighing the impact of each job by the number of date partitions it reads (the orange bars), we see that the majority of IO demand comes from jobs that read 20 or more partitions.

It is also important to understand *which* partitions training jobs typically read. Figure 4 shows a normalized histogram of IO demand over the popular table's date partitions over the course of one day. Partitions do not exhibit flat popularity, but instead show multiple modalities. A large fraction of traffic reads the most recent date partition. This is typical of "recurring" jobs that keep the model up-to-date and exploratory jobs that use the freshest data. There also exist multiple groups of popular date partitions, where multiple larger-scale training jobs use similar date ranges to ensure comparable results.

The above graphs show multiple important characteristics. a) Row reuse is solely across single-epoch training jobs. b) The active working set size of all training jobs is massive. There are over 85 active tables, each containing $O(1-10PB)$ of samples [68]. c) Most IO demand comes from jobs that read tens of date partitions. These jobs have PB-scale working sets due to $O(100TB)$-sized date partitions [68]. There are also many smaller jobs that read a few date partitions. d) Date partitions exhibit varying popularity, with popularity changing over time as date partitions are generated and deleted.

Column-wise Reuse. Columns (i.e., features) also exhibit a distribution in popularity, as training jobs typically use a subset of all features due to hardware (e.g., GPU memory) constraints. Figure 5(a) shows an analysis of 1265 production training jobs that read a specific date partition of the popular

[1]Where relevant, we characterize this same table throughout this section.
[2]A date partition consists of all training samples generated in a given day.

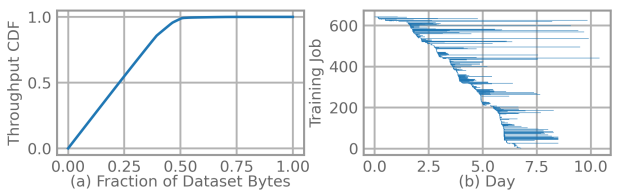


Figure 5: (a) CDF showing distribution of stored feature bytes to IO served to 1265 training jobs reading a single date partition. (b) Time-series of training jobs reading a popular table over one week.

table. The x axis shows a distribution of the stored bytes, ordered by read popularity. The y axis shows the fraction of total IO bandwidth served by most popular x fraction of bytes. Over 75% of bytes are read at least once. However, roughly half of all bytes, corresponding to popular features, serve almost all IO demand.

This has important implications for a cache system, as features are stored into columnar byte streams within large DWRF files. Specifically, row-wise popularity corresponds to *file-popularity*, and column-wise popularity corresponds to *byte-range popularity* within each file. A cache system must effectively capture both dimensions.

Temporal Behavior. Cache systems must also account for the unique temporal characteristics of training jobs. Figure 5(b) shows a time-series plot of 642 jobs, launched over one week, that read the popular table. Each horizontal bar reflects the lifetime of each job, during which it reads the samples and features specified in its dataset. We observe that first, training jobs are largely asynchronous. Cache systems cannot solely optimize for highly synchronized jobs (e.g., hyperparameter tuning) and assume high temporal locality. Secondly, data reuse is expressed across a small number of training jobs, unlike the billions of requests common to web-based workloads [6]. Finally, each training job can run from hours to days, requiring a large storage and temporal footprint.

3.2.2 Cache Software Design Space Exploration

With our workload characteristics in mind, we evaluated various system architectures to manage our flash storage tier.

Option 1: General-purpose Software Caches. We built CacheLib at Meta as a general-purpose cache engine to support caches for datacenter applications such as key-value stores, databases, CDNs, and social graphs [6]. CacheLib offers LRU and FIFO eviction policies over flash, and random and reject first admission policies to manage flash endurance. Our first option was to simply deploy a cluster of flash storage nodes, each managed by CacheLib, to cache file byte ranges.

Unfortunately, ML training workloads exhibit patterns that general-purpose cache policies fail to handle. Our characterization showed a long tail of jobs that read unpopular tables and partitions. However, each job can still have working sets up to tens of petabytes, potentially larger than the entire cache itself. These massive and long-running *scans* can easily evict the entire cache, reducing hits on popular items. Furthermore, even popular training samples are susceptible to *churn*, where each sample is repeatedly evicted and inserted into cache. Churn occurs because a) there are relatively few training jobs in each datacenter, b) data reuse occurs with a relatively long duration between jobs (Figure 5), and c) working set sizes exceed our cache capacities. Scans and churns are well-known antagonist cache patterns [52], motivating the need for specialized and domain-specific admission and eviction policies.

Option 2: ML-specific Caches. We also considered techniques for building an ML-specific cache, inspired by recent work [16, 23, 30, 32, 41, 61, 71], that allows applications to explicitly cache samples in high-bandwidth storage. While such caches can optimize policies for ML workloads, they face several disadvantages. First, current ML caches employ techniques that require assumptions not representative of our workloads, limiting their effectiveness. For example, they cache entire files (e.g., images), and they rely on a large amount of intra-job data reuse across multiple epochs and inter-job data reuse across highly concurrent hyperparameter tuning jobs. Meanwhile, feature popularity requires our cache to store byte ranges within files, and our workloads only exhibit inter-job data reuse with highly asynchronous workloads. Secondly, an application-controlled cache introduces security concerns due to the need to handle access to and deletion of multiple copies of data. Finally, ML caches require end-user efforts to adopt, hindering both our deployment velocity, and more importantly, the productivity of ML engineers.

Option 3: A Transparent, Application-aware Cache. Our final cache design combined benefits from both software and ML caches. We focused on policies that provide the transparency and generalizability of software caches and the application-level awareness of ML caches. Our characterization highlighted key opportunities. Specifically, not only do training jobs tend to favor specific rows and columns, their dataset specifies *which* features, tables, and partitions the job will deterministically read throughout its lifetime.

We next explore how our entire design space exploration yielded a principled design of *Tectonic-Shift*. Section 5 then describes how we leverage policies that infer future access patterns from dataset specifications to minimize cache contention and to maximize the read IO absorbed by the cache.

4 *Tectonic-Shift* Architecture

4.1 *Tectonic-Shift* Design Principles

Figure 6 shows the architecture of *Tectonic-Shift*. We designed *Tectonic-Shift* around four key design principles.

Transparency: Tectonic-Shift combines *Shift*, a flash storage tier, in front of each HDD *Tectonic* cluster transparently. Each *Tectonic-Shift* cluster serves read requests for all training workloads in its respective datacenter. It exposes the same

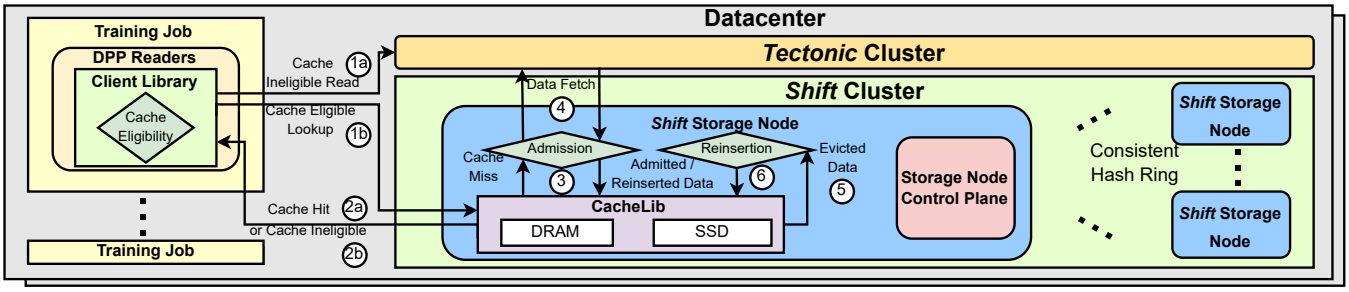


Figure 6: The block diagram of *Tectonic-Shift*, with numbered arrows depicting the path of each read request.

APIs and semantics as our current *Tectonic* File System. Transparency was important for two reasons. First, we avoided exposing storage decisions to ML engineers, as doing so may lead to inefficient configurations and hinder their productivity. It also eased deployment, allowing us to progressively roll out *Shift* and re-balance *Tectonic* clusters under the hood. Secondly, because most of Meta's storage services rely on *Tectonic*, a general API would allow us extend *Shift* to new customers that could become IO-bound (see Section 7).

Simplicity: We kept *Shift* simple and robust by reusing as much infrastructure as possible. *Shift* can only be accessed via the Client Library, which uses *Tectonic*'s Metadata Layer. This simplifies security since all access controls are validated before reaching *Shift*. *Shift* also uses the fact that training data is stored in immutable, *sealed* blocks [50] to avoid managing cache invalidations or mutations. Each *Shift* node uses CacheLib as its internal caching engine, allowing us to harness resources from the myriad teams that rely on CacheLib.

Scalability: We built *Shift* to be effective at any deployment size, quickly deployable, and easily scalable to meet the demands of each datacenter. *Shift* is fully decentralized, consisting of only flash storage nodes placed in a consistent hash ring. Cache decisions are only made locally to each storage node and dynamically adjust based on observed load.

Intelligence: Finally, based on Section 3, *Shift* must adapt to the unique workload characteristics of ML training jobs. Section 5 explores how we built intelligent cache policies on top of CacheLib. These policies infer each training job's data access pattern based on its initial dataset specification, allowing each *Shift* node to maximize IO absorption based on both historic and expected future data access patterns.

4.2 The Life of a *Tectonic-Shift* Read

Client Library. As discussed in Section 2, for each training job, DPP Readers collectively scan through the specified partitions, with each Reader individually reading separate splits of rows. Each partition is mapped to a distinct file system directory, and Readers directly read data corresponding to used features from files in the respective directories. Readers obtain file handles by querying the *Tectonic* Metadata Layer (Figure 2). Mappings from features and rows to file byte ranges are decoded by Readers from file footers. Each Reader issues reads via a `pread` Client Library API call, which returns `count` bytes starting at `offset` within a file.

Figure 6 shows how the Client Library handles each `pread`. It first decomposes the `pread` into a set of block reads by querying the *Tectonic* File Layer. The Client Library then checks if each block read is cache eligible. Cache *in*eligible reads directly read each block range from the *Tectonic* Chunk Store (1a). Cache eligible reads directly issue a `get(blockId, offset, length)` for each block to the *Shift* cluster (1b). We piggyback a number of *tags* with each `get` request that associate each request with relevant metadata, such as the file path and training job ID, to be used by *Shift* policies. The *Shift* cluster consists of a number of flash *Shift* Storage Nodes (SNs) placed in a consistent hash ring [25]. The Client Library maps each `get` request to a *Shift* SN based on `hash(blockId)`. `get` requests either return data for the corresponding block (2a), or return a cache miss (2b). The Client Library reads missed blocks from the *Tectonic* cluster (1a). Once all blocks are fetched, the Client Library returns the results of the `pread` to the caller.

***Shift* Storage Node Data Plane.** *Shift* uses CacheLib [6] within each SN to manage both DRAM and flash. We break up each *Tectonic* block into fixed-size *segments*, which are the objects that we place into CacheLib. Blocks are typically 72 MiB; we discuss segment sizes in Section 5.2.

The SN breaks up each `get` request's range into segments. If all segments are present in cache, the SN simply returns the requested data (2a). Otherwise, *Shift* implements two critical policies on top of CacheLib. First, if any segments are missing, the SN decides if the segment is *admitted* (i.e., allowed) into cache (3). If any segment is not admitted, the SN returns a cache ineligible miss to the Client (2b). Otherwise, the SN will fetch admitted segments from the *Tectonic* cluster and insert them into cache (4). The SN then returns data corresponding to the `get` request (2a). Secondly, any cache insertions will potentially result in segments being evicted from the cache (5). For each evicted segment, the SN can optionally *reinsert* the segment into cache (6), potentially avoiding a cache miss if the evicted segment is accessed in the near future.

***Shift* Abstractions and Guarantees.** The primary goal of

Shift is to serve IO bandwidth corresponding to popular bytes, reducing IO to *Tectonic*'s Chunk Store. We rely on and do not change the semantics of the *Tectonic* File System.

Specifically, the *Tectonic* File System provides append-only semantics. Data pipelines will *seal* blocks; we do not have to handle modifications in *Shift*. *Shift* SNs act only as a part of the data plane. To keep file system operations centralized and scalable, we rely on the *Tectonic* cluster's Metadata Layer for all metadata operations. Thus, *Shift* does not expose a `put` API. Inserts into cache are only made for missed `get` requests. Accesses to blocks are consistent since *Shift* SNs can only be accessed by the *Tectonic* Client Library, which first references the Metadata Layer for file-to-block mappings, preventing reads to renamed or deleted files. Similarly, unauthorized reads are prevented as any the Client Library performs ACL checks for each block before issuing reads to *Shift*.

Our proposed design also allows *Shift* to be inherently fault tolerant. *Shift* contains no centralized state. *Shift* relies on the fault tolerant *Tectonic* Metadata Layer [50] for metadata operations. Cache policies are made local to each SN, allowing other SNs to proceed unhindered in the event of a SN failure. Furthermore, *Shift* serves only reads, and Clients will default to *Tectonic* reads (e.g., after a timeout) if a given SN fails.

5 Application-Aware Cache Policies

Building on our design principles, our key insight is to instrument *Shift* with intelligent policies that maximize the IO absorbed from *Tectonic*. These policies transparently adapt to workloads by leveraging both historic and application information, ensuring that only segments with high reuse across training jobs are kept in each SN. Specifically, each *Shift* SN contains a Control Plane that implements an **admission** and **reinsertion** policy. We incorporate a **cache eligibility** policy in the Client Library to reduce RPC pressure and avoid requests to *Shift* from "uncacheable" workloads.

We focused on building a flexible SN Control Plane that aggregates the necessary metadata (including application information) to define and inform highly configurable policies, allowing us to constantly tune and improve performance. We prioritized admission and reinsertion policies as opposed to eviction policies such as LRU because a) we can prevent significant thrashing due to the scan and churn (Section 3) patterns common in our workloads, and b) we can control write rates to flash in order to manage flash endurance constraints. Meanwhile, our policies are built on top of CacheLib, and we use CacheLib's provided eviction policies (LRU or FIFO) after admission into cache.

As discussed in Section 4, each *Shift* SN acts as an independent entity, serving only requests on its portion of a consistent hash ring. Thus, the overall goal of *each Shift* SN is to maximize the absorbed IO locally — doing so maximizes the aggregate IO absorbed by the entire *Shift* cluster. We define the absorbed IO at each SN as the bandwidth of successful

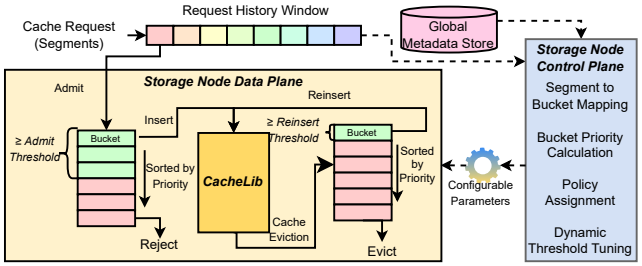


Figure 7: Overview of *Shift* SN Control and Data Plane. The Control Plane dynamically tunes Data Plane cache policies.

Table 2: Table listing configurable parameters in *Shift*.

Configurable Parameter	Meaning
$MapSegment(s)$	Mapping function from segment to bucket.
$BucketPriority(b)$	Function to calculate priority of bucket b.
$AdmitThreshold$	Dynamic scalar threshold to admit buckets.
$ReinsertThreshold$	Dynamic scalar threshold to reinsert buckets.
$BucketRefreshTime$	Update period for bucket policies.
$RHWSize$	Size of request history window logs to keep.

`get` requests returned to clients, minus the bandwidth of data fetches to the *Tectonic* cluster due to data misses. Importantly, non-admitted cache requests do not impact absorbed IO, and reinserted segments do not contribute to fetches to *Tectonic*.

5.1 Admission and Reinsertion

Figure 7 shows an overview of the Control and Data Plane running in each SN. The Control Plane dynamically directs the Data Plane to admit or reinsert segments into the cache upon a cache miss or eviction, respectively, using *historical* and *application* metadata. These decisions are made based on the configurable parameters summarized in Table 2.

Mapping a Segment to a Bucket. The Control Plane statically maps each segment s to a *bucket* b using the tags attached to each read request, based on a configurable function $MapSegment(s)$. Intuitively, each bucket represents a collection of segments that will likely be accessed together and connects to a logical grouping within the application.

For example, we typically use a segment's corresponding directory as its default bucket mapping, as each training job specifies a set of partitions to read and will scan through files within each partition's directory. Furthermore, we can correlate data reuse across multiple jobs based on their dataset partitions (and thus directories). While finer-grained bucket mappings such as files or features (i.e., file byte ranges) are possible, directories provide sufficient granularity since jobs mostly read similar features within each file (Section 3).

Assigning a Policy to Each Bucket. Each bucket is assigned a binary admission and reinsertion policy. The Data Plane simply admits/rejects (on miss) or reinserts/evicts (on eviction) each segment depending on its bucket's current policy. The Control Plane will admit or reinsert a bucket if the bucket's current *BucketPriority(b)* is greater than *AdmitThreshold* or

ReinsertThreshold, respectively. Bucket policy assignments are updated every *BucketRefreshTime*, a configurable parameter. Buckets should be updated frequently enough to react to changes in reading patterns; we use a default of 10 seconds.

Deriving a Bucket's Priority. *BucketPriority(b)* directly determines b's admission/reinsertion policy. Intuitively, we interpret b's priority as *the number of times we expect each segment in the bucket to be read in the near future*. Higher priority buckets will be placed into the cache (via admission/reinsertion) over lower-priority buckets and thus absorb more IO in total. Our key insight is to calculate buckets' priorities using both *historical* and *future* information derived from the Request History Window (RHW) and Global Metadata Store (GMS), respectively.

Historic Priority. The RHW tracks recently observed requests (regardless of admission) over a past time period, $RHWSize$. The RHW reports the number of *unique* segments and *total* segments requested for each bucket. A *historic priority* for bucket b can be calculated as $BucketPriority(b) = TotalBytes(b)/UniqueBytes(b)$, providing the traditional cache signal which assumes that past access patterns are indicative of the future. $RHWSize$ is a configurable parameter. We tune it to capture sufficient historical data without exceeding memory capacity limits; we use a default of 6 hours.

Future Priority. The RHW also records the set of active training jobs and the set of buckets read by each training job using the job ID tag piggybacked with each request. The GMS is a set of databases which contains real-time information about the dataset specification of each training job. By combining the RHW and GMS, *we can directly derive future accesses for each training job* and thus bucket priorities. The Control Plane queries the RHW for all active training jobs and pulls each job's dataset specification from the GMS. For example, for directory-based buckets, the Control Plane derives a bucket's *future priority* as equal to the number of jobs that include the corresponding partition in its dataset, discounting any jobs that have finished reading the bucket.

In summary, the RHW captures historic access patterns and active training jobs, while the GMS associates each training job with application information about its dataset. While we presented potential historic and future policies, variations can easily be created using the RHW and GMS. For example, a potential future policy can further prioritize directories earlier in read order (and thus read sooner). Section 6 explores a *hybrid* policy combining historic and future priorities.

Threshold Tuning. The Control Plane continuously tunes the $AdmitThreshold$ and $ReinsertThreshold$ based on two factors. First, a *minimum threshold* avoids admitting unpopular workloads that can evict the entire cache. We use a minimum value that is strictly greater than 1, and we constantly tune it based on observed performance. Secondly, we implement a PID-controlled *feedback loop* to ensure that the cache admit plus reinsert rates (reported by the RHW) is strictly less than our flash endurance limits (defined by a maximum average write rate). This allows the threshold to increase beyond the minimum in response to high flash write rates, thereby admitting/reinserting fewer segments. We only tune the $AdmitThreshold$ and tie the $ReinsertThreshold$ to be a fixed offset (e.g., $AdmitThreshold + 1$) to prioritize admits over reinserts to limit write amplification due to reinserts.

An additional benefit in building an admission policy *above* CacheLib is to rate limit prior to *Tectonic* reads. While CacheLib provides a rate limiter, which selectively admits segments to flash upon DRAM eviction, it inherently requires first inserting data into DRAM. This results in unnecessary *Tectonic* reads if the data is soon to be evicted due to write endurance limits. *Shift*'s admission policy acts prior to *Tectonic* fetches, avoiding unnecessary HDD reads for rate limited data.

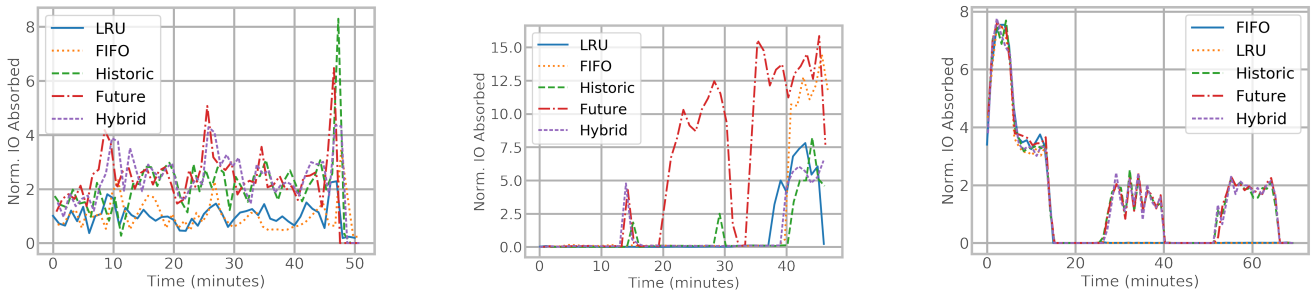
5.2 CacheLib Tuning

CacheLib offers a suite of configurable parameters that we continuously tune via a host of stress, release validation, and production tests. While prior flash caches focused on addressing write amplification caused by small objects (e.g., <1KB messages) [6, 13, 38, 56], a key difference and advantage in *Shift* is the ability to configure segment sizes. Too large segments, relative to request sizes, result in overheads since we fetch and store data in segment-granularity. On the other hand, too small segment sizes constrain both DRAM and flash due to metadata and write amplification overheads. We found that 256 KB was a good balance for our workloads. We also rely on CacheLib to optimize underlying data layouts on flash [6] to further improve flash endurance. Finally, we evaluated the available eviction policies for DRAM and flash and found that LRU works well (when combined with *Shift*'s policies); we provide further exploration in Section 6.

5.3 Client Cache Eligibility

We also incorporate a *Shift* eligibility policy at the *Tectonic* Client Library. *Tectonic* serves a diverse set of training and non-ML workloads across Meta. The primary purpose of the cache eligibility policy is to prevent customers that are not onboarded to *Shift*, as well as low-priority and uncommon (and less-cacheable) tables, from issuing lookups to *Shift*. While these read requests would likely be rejected by the SN's cache admission policy, applying a first filter significantly reduces the RPC load and memory pressure at each SN.

We currently filter out all non-ML traffic to bolster ML training capacity. Furthermore, Figure 3 shows that tables are disproportionately popular. Filtering out rarely-used tables adds another layer of protection against cache contention, since all IO can be sufficiently served from *Tectonic* HDDs. As a baseline heuristic, we filter out tables whose IO demand can be sufficiently served by the HDD capacity needed to store it. We continuously tune our filters based on demand.



(a) Synchronized workload (b) Pipelined workload (c) Sequential workload

Figure 8: IO absorbed by different policies across benchmarks, normalized to the average IO absorbed by *LRU* eviction.

Table 3: Benchmark DLRM training job workloads used for evaluation. Each job j is denoted as $\{\}_j$ and reads each partition P in specified order. Each partition P has a $\approx$ 5 TB working set.

Workload	Jobs & Partitions Read	Description
Synchronized	$\{P_1,P_2,P_3\}_1$, $\{P_4,P_5,P_6\}_2$, $\{P_1,P_2,P_3\}_3$, $\{P_7,P_8,P_9\}_4$, $\{P_1,P_2,P_3\}_5$	Multi-tenant HP tuning or exploratory jobs. Jobs are launched synchronously.
Pipelined	$\{P_1,P_2,P_3\}_1$, $\{P_2,P_3\}_2$, $\{P_3\}_3$	Long-running, pipelined jobs. Jobs are launched synchronously.
Sequential	$\{P_1\}_1$, $\{P_1\}_2$, $\{P_1\}_3$, $\{P_1\}_4$, $\{P_2\}_5$, $\{P_3\}_6$, $\{P_1\}_7$, $\{P_4\}_8$, $\{P_5\}_9$	Queued jobs that launch when training capacity is available. Jobs 1-3, 4-6, and 7-9 launch together.

6 *Tectonic-Shift* Deployment and Evaluation

Tectonic is Meta's durable storage system. It has been in production since 2015 and stores exabytes of data. *Shift* has been in production since early 2022 and is deployed at petabyte-scale alongside multiple *Tectonic* clusters serving DLRM training workloads. In this section, we evaluate *Shift*'s various caching policies compared to state of the art using a series of representative workloads, and we present results of *Shift* in production. We focus on and report absorbed IO because it is *Shift*'s top-line metric and optimization goal. A comparison of hit rates would yield analogous results since requests rates are equally distributed across SNs due to consistent hashing.

6.1 *Shift* Policy Evaluation

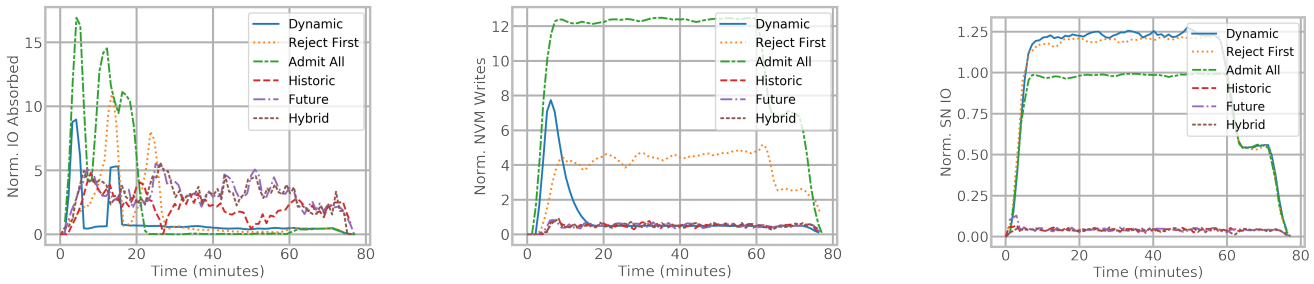
To better understand how *Shift* policies perform, we used a set of representative workload patterns shown in Table 3. Each pattern was derived from production DLRM training job traces and downsized to scale to our evaluation cluster. The *Synchronized* pattern represents a case where multiple training jobs reading the same partitions are launched at the same time (e.g., for hyperparameter tuning jobs), with training jobs reading other partitions interleaved due to the multi-tenancy of our training clusters. The *Pipelined* pattern frequently occurs in long-running jobs when users kill an under-performing job and replace it with a new model using the same dataset. The *Sequential* pattern occurs due to limited training resources, where jobs will run in separate batches as jobs finish and resources become available. For each workload, we used a set of Readers that each read from *Tectonic-Shift* at $\approx$ 1.5 GB/s.

We evaluated on a 6-node *Shift* cluster deployed with our production configuration. In each experiment, we configured nodes with different policies and evaluated each policy concurrently to ensure equal read locality; consistent hashing evenly spread requests across nodes. We used 16 GB of DRAM cache for each node. The Synchronized, Pipelined, and Sequential patterns used (1.28 TB, 5 TB, and 5 TB), and (4, 5, and 5) of total flash cache and Readers per job, respectively. Unless otherwise stated, we used directory bucket mappings and disabled reinsertion and write rate limits.

Do admission policies improve IO absorption? First, we evaluated if various *Shift* admission policies improved IO absorption across all workloads. We used the *Historic* and *Future* admission policies presented in Section 5.1, which use historic and future metadata, respectively, from the RHW and GMS to calculate a priority equal to the number of expected reads per segment. We also used a *Hybrid* admission that uses $BucketPriority_{Hybrid}(b) = max(BucketPriority_{Historic}(b), BucketPriority_{Future}(b))$. We set a minimum admit threshold of 1.1 for each policy, implementing a "reject first" policy. We used LRU eviction for each admission policy, and we compared to two baselines that only used CacheLib's *FIFO* and *LRU* eviction policies.

Figure 8 shows the IO (bandwidth) absorbed by each policy, normalized to the average IO absorbed by *LRU*. A higher IO absorption directly translates to higher *Shift* efficiency. In the Synchronized workload, *FIFO* absorbed an equal amount of IO (1.01×) as *LRU*. The *Historic* policy absorbed 2.01× more IO than *LRU*, since it was able to avoid cache thrashing induced by P_{4-9} (see Table 3) by not admitting them. The *Future* and *Hybrid* policies absorbed 2.27× and 2.32× more IO, out-performing *Historic* admission since they were also able to immediately cache $P_{1,2,3}$ without rejecting initial reads waiting for the Request History Window to populate.

For the Pipelined workload, *FIFO* was able to absorb 1.86× more IO than *LRU*, since churn caused *LRU* to evict more objects in P_3 before job 1 read P_3. *Historic* admission performed worse than *LRU*, absorbing only 0.80× IO, as it did not admit any bytes from P_3 until job 2. Since the *Future* admission policy immediately knew of P_2 and P_3's popularity,



(a) IO Absorbed (b) NVM Write Rate (c) HDD Read / Cache Fill Rate

Figure 9: Policy performance using the Synchronized workload, normalized to *Dynamic*. *Dynamic* and *Shift* write limits are set to 100 MB/s.

it was able to maximize and absorb 5.84× more IO than *LRU* by admitting them to both to cache on the first read by jobs 2 and 3. *Hybrid* admission equalled *LRU* (0.99×), since it kept admitting P_2 during job 1, expecting future reads due to historic popularity and thrashing P_3's data in cache.

Finally, we observe that for the Sequential workload, *FIFO* performed on-par (1.06× IO absorbed) with *LRU*, while *Historic*, *Future*, and *Hybrid* outperformed *LRU* equally (1.71×, 1.74× and 1.69× respectively). Specifically, for the first set of jobs (1-3), all policies saw high hit rates since only P_1 was actively read. However, for the second (4-6) and third (7-9) sets of jobs, the *Shift* policies rejected reads from P_{2-5}, avoiding contention and improving IO absorbed by P_1.

On average across all workloads, *FIFO*, *Historic*, *Future*, and *Hybrid* respectively absorbed 1.31×, 1.51×, 3.28×, and 1.67× more IO compared to *LRU*.

Can admission policies manage flash endurance? We need to limit the write rate to SSDs in order to preserve their lifetime. To study how well *Shift* policies perform under constrained write limits, we repeated the Synchronized workload while limiting each *Shift* node with an flash write limit. We compared to two state-of-the-art admission policies provided by CacheLib: a *Dynamic* flash admission policy randomly rejects writes to flash in order to maintain the specified write limit, and a *Reject First* flash admission policy rejects objects' first write to flash. We also evaluated no admission policy (*Admit All*). We used LRU eviction for all admission policies, and we configured *Shift* policies and the *Dynamic* policy to write only 100 MB/s per node. To fully evaluate the *Shift* threshold tuner, we did not set a minimum admit threshold.

Figure 9 shows the IO absorbed, flash write rate, and *Tectonic* HDD read rate for each admission policy, with each metric normalized to the average for the *Dynamic* admission policy. All admission policies absorb more IO than *Dynamic*. *Reject First* and *Admit All* absorb 1.51× and 2.66× more IO, respectively; the *Historic*, *Future*, and *Hybrid* policies absorb 2.14×, 3.07×, and 2.99× more IO, respectively.

While *Shift*'s *Future* and *Hybrid* policies performed similarly to *Admit All*, Figure 9b shows how *Admit All* required significantly more (10.38×) flash writes than *Dynamic*, exceeding our write limit. *Reject First* was similarly ineffective, requiring 3.78× more flash writes. Meanwhile, all of *Shift*'s policies matched CacheLib's write rate (within 5%, even discounting CacheLib's increased writes initially due to its counter warm-up) while outperforming its IO absorption.

Table 4: Hit rate and HDD IO (cache fills) using *Hybrid* admission with dynamically-tuned reinsertion, normalized to *Hybrid* admission without reinsertion. Write rate is limited to 1 GB/s.

	Normalized Hit Rate	Normalized HDD IO
Hybrid with Reinsertion	1.03	0.82

Furthermore, *Shift* has the advantage of avoiding excess reads from *Tectonic* HDD nodes for rejected objects, compared to CacheLib's *Dynamic* policy which always admits objects to DRAM first (and thus incurring an HDD read) before rejecting it from flash. Figure 9c shows this benefit; each of *Shift*'s policies avoids an HDD read upon rejection, significantly reducing the amount of cache fills (96% less than *Dynamic*) compared to CacheLib's baseline policies.

These results show that *Shift*'s threshold tuning mechanism is effective at maximizing IO absorption given a write constraint, without incurring excess *Tectonic* cluster reads.

How effective is reinsertion? We evaluated if reinsertion was effective at reducing HDD reads compared to admission only. We repeated the Synchronized workload with a write limit of 1 GB/s and a reinsertion threshold 1.0 greater than the dynamic admission threshold with a minimum admit of 1.1. We compared a *Hybrid* admission policy with reinsertion against a baseline *Hybrid* policy without reinsertion.

Table 4 shows the hit rate and HDD IO with reinsertion enabled, normalized to the baseline without reinsertion. We observe that enabling reinsertion resulted in similar hit rates (3% increase), while reducing HDD reads by 18%. However, compared to the limited flash write rate of the baseline (≈ 300 MB/s), enabling reinsertion with dynamic threshold tuning resulted in *Shift* always hitting the write limit, since reinsertions caused more reinsertions until the limit was exceeded. Our takeaway is that currently, reinsertions may be effective when reducing HDD reads are prioritized over reducing flash writes. However, potential future optimizations in CacheLib (see Section 7) can harness reinsertion's benefits while eliminating the write overheads caused by continued reinsertions.

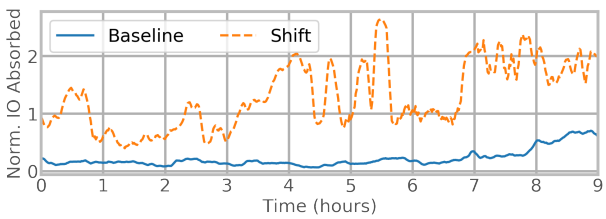


Figure 10: Production results comparing *Shift* to an expert manual-tuned policy that admits only IO-heavy tables.

6.2 Production Results

We have enabled the policies presented in Section 5 in our production clusters. Since the mix of training jobs and resources varies across datacenters, we are continuously tuning policy configurations for each cluster. These policies have helped *Shift* save significant amounts of power.

To demonstrate this, Figure 10 shows a representative trace of the IO absorbed by *Shift* nodes in a petabyte-scale production cluster over the course of 9 hours. We show a baseline that uses an *Expert*-tuned admission policy on top of LRU, which admitted only high-IOPS tables to cache; no admission policy (admit all) showed near-zero IO absorption due to significant cache contention across training jobs. We compared against *Shift* using a *Hybrid* policy, with a minimum admit threshold of 3.0 and without reinsertion. Both used client eligibility policies and our production write rate limit (the baseline additionally used *Dynamic* admission if necessary), and we normalized results to the "power-neutral" IO absorption point: the required amount of IO *Shift* needs to absorb to reduce power consumption compared to using only HDDs.

The *Expert* admission policy is ineffective at saving power due to its inability to capture the limited data reuse of training jobs we characterized in Section 3. Simply deploying a flash cache without intelligent and adaptable cache policies is inefficient; our production trace shows that doing so would only absorb $0.21\times$ the IO needed to achieve power neutrality. By employing the application-aware policies presented in Section 5, we show that *Shift* can exploit the unique characteristics of training jobs, saving 29% of power relative to using only HDDs for training data storage. At our scale, this corresponds to a massive efficiency improvement.

7 Lessons Learned and Open Questions

Define the right interface to users. Our focus on designing a *transparent* but *intelligent* interface to users was instrumental in the success of *Tectonic-Shift*. A transparent interface allowed us to quickly onboard new users by simply configuring the cache eligibility policy, requiring *zero* application modifications. Since applications could be agnostic to their use of *Shift*, this allowed us to dynamically manage the deployment and operation of *Shift* with fine granularity. For example, we could gradually roll-out to a new customer, A/B test different policies across SNs, or even roll-back to reduce request pressure — all without affecting customers' performance.

At the same time, we quickly learned that it was essential to work closely with customers to maximize *Shift* performance. We collaborated heavily with AI infrastructure and ML engineering teams to understand and extract the right set of application metadata to optimize *Shift* policies. Looking forward, we believe that a wide range of other ML domains (e.g., vision, NLP, etc.) and non-ML applications will benefit from *Shift*. Since these applications use the same *Tectonic* API, our focus can simply lie in working with customers to extracting the best features to maximize *Tectonic-Shift* efficiency.

Rely on a slate of robust testing, experimentation, and monitoring mechanisms. We use multiple tools such as stress tests, production A/B testing, trace-based simulation, shadowing, and dashboards to continuously tune the multiple *Shift* policies and configurations discussed in Section 5. These tools have also helped ensure a stable deployment of *Shift*. For example, data corruptions are common at our scale, requiring us to compute, store, and verify checksums for each *Shift* segment. To ensure data correctness prior to deployment, we used a shadow deployment to have clients fetch *Shift* checksums to compare against data read from *Tectonic*.

Effectively use DRAM. CacheLib uses both DRAM and flash to store data. For our use case, DRAM capacity was negligible relative to the orders of magnitude larger flash capacity. Instead, we prioritized using DRAM to store metadata (e.g., the RHW) to improve the effectiveness of *Shift* policies, but we still had to reserve tens of gigabytes of memory for CacheLib to buffer and serve requests at high throughput. Further CacheLib optimizations to reduce memory requirements and an investigation into the optimal split between data and metadata may further improve *Shift* performance.

Can data placement and job routing policies improve cache performance? A key opportunity we foresee is co-designing *Tectonic-Shift* with data placement and training job routing policies. Data placement policies govern how tables are replicated across our datacenters. Cache-aware data placement policies can help reduce cache working set sizes by intelligently reducing data replication while balancing for data availability. Cache-aware job routing policies can improve the IO absorbed by *Shift* by coalescing jobs that read similar data within the same datacenter.

How much can priority-aware evictions improve cache performance? Section 6 showed that while reinsertion was effective, it required significant flash writes due to reinsertion cycles. Reinsertions are necessary because CacheLib only offers LRU or FIFO eviction. We believe that further optimizations in CacheLib, such as allowing selective evictions based on a cache object's priority, can harness the benefits of reinsertion without write overheads.

Can historic and future knowledge inform better cache policies? *Shift* provides an extensible framework to build cache admission and reinsertion policies based historic and

future information. While we demonstrated that a hybrid policy based on the maximum of historic and future priorities was effective, a promising research direction is to explore novel cache policies that can leverage future information. For example, a potential cache policy may account for *when* objects will be read in the future and prioritize earlier objects.

8 Related Work

Software Flash/DRAM Caches. Systems such as Redis [36], memcached [45], RAMCloud [49], and Pocket [29] are widely used as caches across datacenter applications. These software caches commonly manage DRAM and/or flash using a mix of policies including admission (e.g., TinyFLU [12] and LARC [20]) and eviction/replacement policies (e.g., ARC [39], LRU [47], 2Q [22], and OPT [5]) that leverage historical (or oracular) access patterns. Techniques to predict file access patterns, e.g., access trees [33], are also well studied.

Recent works have also proposed mechanisms to dynamically tune these policies using ML models [34, 52, 59], hardware access signatures [63], NLP techniques [17, 69], and other heuristics [40, 66]. Meanwhile, flash-specific policies [6, 13, 38, 56, 65] largely focus on managing flash write amplification (WA). CacheSack [66] is used as the admission policy for Google's Colossus Flash Cache, which shares a similar goal to *Shift* in absorbing IO from HDDs. CacheSack splits objects by category and tunes the admission policy of each category. Janus [4] is also a flash tier used in Google's Colossus file system, but instead requires files to be written to flash first before being evicted to HDDs.

Various existing storage systems also leverage architectures similar to *Tectonic-Shift*. Swift uses a set of configurable hash rings to map objects to their respective storage device [48]. Numerous file systems leverage heterogeneous storage devices to optimize for performance and efficiency across various applications [24, 60, 62, 64]. For example, burst buffers, typically consisting of IO-performant devices such as flash, are commonly used to absorb peaks of high IO-demand from backend (e.g., HDD) storage systems in high-performance computing applications [35].

Tectonic-Shift is a composite storage fabric that employs a mix of cache policies across *Tectonic* Clients and *Shift* Storage Nodes to maximize its efficiency. *Shift* runs a CacheLib [6] instance in each SN. We use comparatively large segments (256 KB) and rely on CacheLib's Large Object Cache to handle WA. Each *Shift* SN dynamically tunes its cache policies, including admission and reinsertion, independent of CacheLib's. While *Tectonic-Shift* shares a similar goal with Google's CacheSack [66] and Janus [4], and leverages well-known techniques such as consistent hashing and heterogeneous devices, it uniquely targets industrial ML training jobs and adopts novel application-aware policies that infer future access patterns from job specifications.

ML-specific Caches. Recent work has shown the utility of caches for ML training workloads. CoorDL [41] eliminates data stalls in single-server training using local SSDs. Quiver [30] and OneAccess [23] cache and share data across highly-synchronized HP tuning jobs. DIESEL [61] targets training workloads over small files (e.g., images). DLFS [71] and DeepIO [70] randomize mini-batches using specialized hardware. Cachew [16] builds on tf.data [44], and puts intermediate data in cloud storage to reduce preprocessing costs.

While other ML caches require adoption effort, *Tectonic-Shift* is completely transparent to users. Furthermore, Section 3 explored why industrial ML training workloads present novel challenges not addressed by these caches. *Tectonic-Shift* is designed for exascale and continuously serves traffic to datacenter-scale GPU training clusters.

Production ML Workload Characterization. tf.data [44] provided an ML training workload characterization at Google, highlighting similar traits such as prevalent data reuse and selective reading. *Tectonic-Shift* is motivated by and builds upon prior characterization of Meta's training workloads [68].

Distributed File Systems. *Tectonic-Shift* is built on top of *Tectonic* [50] and provides the same API and append-only semantics as the *Tectonic* File System. Other distributed file systems, such as Spanner [10], GFS [15], Colossus [19], HDFS [54], and Lustre [53], are used across industry.

9 Conclusion

We presented *Tectonic-Shift*, the composite storage fabric used in Meta's production ML training infrastructure. *Tectonic-Shift* maximizes efficiency by balancing storage and IO capacity across HDD and flash. We provided an in-depth workload characterization and design space exploration which guided the principled design of *Tectonic-Shift*. *Shift* employs a set of application-aware policies that infer and exploit future access patterns using job specifications. We demonstrated how *Shift* absorbed $1.51-3.28\times$ more IO than an LRU flash cache and improved the power efficiency of *Tectonic-Shift* by 29%.

Acknowledgments

We are grateful to the anonymous reviewers and to our shepherd, Apoorve Mohan, whose comments have greatly helped improve this paper. We would also like to acknowledge the contributions of Rocky Wang, Mario Consuegra, Cem Cayiroglu, Jolene Tan, and many others at Meta who have played a vital role in this endeavor. Christos Kozyrakis was partially supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Mark Zhao was supported by a Stanford Graduate Fellowship while at Stanford University.

References

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. Scuba: Diving into data at facebook. *Proc. VLDB Endow.*, 6(11):1057–1067, aug 2013.

[2] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814, Los Alamitos, CA, USA, mar 2021. IEEE Computer Society.

[3] Jason Adrian. Introducing bryce canyon: Our next-generation storage platform. https://engineering.fb.com/2017/03/08/data-center-engineering/introducing-bryce-canyon-our-next-generation-storage-platform/, 2017.

[4] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C. Eric Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 91–102, San Jose, CA, June 2013. USENIX Association.

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020.

[7] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 483–498, USA, 2017. USENIX Association.

[8] Matt Bowman, Abe Garcia, Jun Shen, Haken Michael, Wei Zhang, and Ross Stenfort. Yosemite v3: Sierra point e1.s 2ou flash blade and expansion board design specification. https://www.opencompute.org/documents/e1s-expansion-2ou-1s-server-design-specification-pdf, 2021.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, June 2013. USENIX Association.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ 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 *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[11] Western Digital. Wd gold ssd product brief. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/wd-gold-ssd/product-brief-wd-gold-enterprise-class-nvme-ssd.pdf, 2022.

[12] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 146–153, 2014.

[13] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, Boston, MA, February 2019. USENIX Association.

[14] Apache Software Foundation. Apache orc: High-performance columnar storage for hadoop. https://orc.apache.org/, 2022.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 29–43, New York, NY, USA, 2003. Association for Computing Machinery.

[16] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference*

(USENIX ATC 22), pages 689–706, Carlsbad, CA, July 2022. USENIX Association.

[17] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1919–1928. PMLR, 10–15 Jul 2018.

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[19] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system/, April 2021.

[20] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with lazy adaptive replacement. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2013.

[21] Facebook Inc. Hive-dwrf. https://github.com/facebookarchive/hive-dwrf, 2015.

[22] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[23] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[24] Elena Kakoulli and Herodotos Herodotou. Octopusfs: A distributed file system with tiered storage management. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 65–78, New York, NY, USA, 2017. Association for Computing Machinery.

[25] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

[26] Manolis Karpathiotakis, Dino Wernli, and Milos Stojanovic. Scribe: Transporting petabytes per hour via a distributed, buffered queueing system. https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/, Oct 2019.

[27] Andrej Karpathy. Software 2.0. https://karpathy.medium.com/software-2-0-a64152b37c35, Mar 2021.

[28] 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, New York, NY, USA, 2016. Association for Computing Machinery.

[29] 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 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[30] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.

[31] Frederic Lardinois. Google launches a 9 exaflop cluster of cloud TPU v4 pods into public preview. https://techcrunch.com/2022/05/11/google-launches-a-9-exaflop-cluster-of-cloud-tpu-v4-pods-into-public-preview/, May 2022.

[32] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 537–550. USENIX Association, July 2021.

[33] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *USENIX 1997 Annual Technical Conference (USENIX ATC 97)*, Anaheim, CA, January 1997. USENIX Association.

[34] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[35] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2012.

[36] Redis Ltd. Redis. https://redis.io/, 2022.

[37] Sarang Masti. How we built a general purpose key value store for facebook with zippydb. https://engineering.fb.com/2021/08/06/core-data/zippydb/, Aug 2021.

[38] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.

[40] Michael P. Mesnier and Jason B. Akers. Differentiated storage services. *SIGOPS Oper. Syst. Rev.*, 45(1):45–53, feb 2011.

[41] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proc. VLDB Endow.*, 14(5):771–784, jan 2021.

[42] 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. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 993–1011, New York, NY, USA, 2022. Association for Computing Machinery.

[43] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, October 2014. USENIX Association.

[44] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[45] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[46] Kunle Olukotun. Designing computer systems for software 2.0. https://iscaconf.org/isca2018/docs/Kunle-ISCA-Keynote-2018.pdf, June 2018.

[47] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[48] Openstack. The rings. https://docs.openstack.org/swift/latest/overview_ring.html, 2023.

[49] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.

[50] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing,

Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[51] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[52] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354, 2021.

[53] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[54] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[55] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[56] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, Santa Clara, CA, February 2015. USENIX Association.

[57] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[58] Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. Shared foundations: Modernizing meta's data lakehouse. In *The Conference on Innovative Data Systems Research, CIDR*, 2023.

[59] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[60] Lukas Vogel, Viktor Leis, Alexander van Renen, Thomas Neumann, Satoshi Imamura, and Alfons Kemper. Mosaic: A budget-conscious storage engine for relational database systems. *Proc. VLDB Endow.*, 13(12):2662–2675, jul 2020.

[61] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[62] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. *ACM Trans. Comput. Syst.*, 14(1):108–136, feb 1996.

[63] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 430–441, New York, NY, USA, 2011. Association for Computing Machinery.

[64] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, feb 2019.

[65] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. Hec: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[66] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association.

[67] 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 *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

[68] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra 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 data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 350–364, 2021.

[70] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156, 2018.

[71] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. Efficient user-level storage disaggregation for deep learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019.

Calcspar: A Contract-Aware LSM Store for Cloud Storage with Low Latency Spikes

Yuanhui Zhou[1], Jian Zhou[1]✉, Shuning Chen[2], Peng Xu[3]✉, Peng Wu[1], Yanguang Wang[2], Xian Liu[2], Ling Zhan[4], Jiguang Wan[1]

[1] WNLO, Huazhong University of Science and Technology, Wuhan, Hubei, China
[2] PingCAP, China
[3] Research Center for Graph Computing, Zhejiang Lab, Hangzhou, Zhejiang, China
[4] Division of Information Science and Technology, Wenhua University, Wuhan, China
✉Corresponding author {jianzhou@hust.edu.cn, xup@zhejianglab.com}

Abstract

Cloud storage is gaining popularity because of features such as pay-as-you-go that significantly reduces storage costs. However, the community has not sufficiently explored its contract model and latency characteristics. As LSM-Tree-based key-value stores (LSM stores) become the building block for numerous cloud applications, how cloud storage would impact the performance of key-value accesses is vital. This study reveals the significant latency variances of Amazon Elastic Block Store (EBS) under various I/O pressures, which challenges LSM store read performance on cloud storage. To reduce the corresponding tail latency, we propose Calcspar, a contract-aware LSM store for cloud storage, which efficiently addresses the challenges by regulating the rate of I/O requests to cloud storage and absorbing surplus I/O requests with the data cache. We specifically developed a fluctuation-aware cache to lower the high latency brought on by workload fluctuations. Additionally, we build a congestion-aware IOPS allocator to reduce the impact of LSM store internal operations on read latency. We evaluated Calcspar on EBS with different real-world workloads and compared it to the cutting-edge LSM stores. The results show that Calcspar can significantly reduce tail latency while maintaining regular read and write performance, keeping the 99th percentile latency under 550μs and reducing average latency by 66%.

1 Introduction

In recent years, the trend that many businesses and organizations shift their data to the cloud has fueled the growth of cloud storage [21, 34]. This is due to the advanced features and cost-effectiveness of cloud storage. For example, Amazon Web Services (AWS), the world's most broadly adopted cloud platform, provides various storage services with high scalability and reliability on a pay-as-you-go basis [8] (e.g., Elastic Block Store, EBS), making them more appealing. Another important trend is that LSM-Tree-based key-value stores (LSM stores), such as RocksDB [5], LevelDB [1], Bigtable [14], Dynamo [17] and TiDB [19], are becoming the building block for many cloud applications. However, none of the existing LSM stores is optimized for cloud storage to eliminate long-tail latency. Notably, it is challenging to balance the estimated peak performance with the budget for cloud storage performance (e.g., IOPS). Although many cloud storage providers advertise elastic storage volumes that can accommodate changing performance needs, these volumes' scaling capabilities fail to adapt to an inevitable traffic fluctuation. For instance, AWS EBS only supports increasing the purchased IOPS, which would take hours or even days to complete [2]. Hence, it is impractical to rely solely on elastic volumes for timely adjustments in the face of short-term workload changes.

To understand how cloud storage would respond to traffic fluctuation, we have explored the latency characteristics of AWS EBS volumes. Results show that EBS guarantees a service agreement called Service Level Agreement (SLA) in which the processing latency of each request falls within an appropriate threshold *if* the accesses do not exceed the paid IOPS. We observe that the processing latency of each consecutive request dramatically increases when the demands in a time window exceed the IOPS agreement. Besides, the cloud storage's contract model shows that the higher the paid IOPS, the lower the latency. However, such an agreement is constrained by IOPS budgets, and naturally, performance in terms of latency will suffer if the IOPS is overdrawn.

The latency spike caused by limited IOPS in cloud storage severely impacts the performance of latency-sensitive applications on top of LSM stores. We take one of the most widely deployed LSM store implementations, RocksDB, as an example. The RocksDB first writes the in-memory table (memtable) to respond quickly with reasonably low latency. Until the in-memory table is full, RocksDB then persists the table to the storage volume in large chunk writes (e.g., SSTables), thus aggregating the random writes into sequential ones. Such a write scheme reduces the number of write requests and achieves high write throughput. It then employs internal compaction mechanisms to merge and resort the incoming data

with multiple levels of on-disk tables. Although the internal compaction operation ensures the orderliness of data in each level to improve the lookup performance, a read operation still needs to traverse multiple levels, resulting in read amplification. As the IOPS on a cloud storage volume is limited, the read performance of RocksDB is significantly throttled.

There are several challenges to avoid read latency spikes in LSM stores. First, the read request performance fluctuates significantly because the cloud storage volume isn't flexible enough to timely keep up with the changing workload. The fluctuating workload causes the number of read I/Os of an LSM store to access cloud storage volumes to vary significantly. The request latency increases when the I/O number exceeds the paid IOPS of the cloud storage volume. Second, the read amplification in an LSM store further strengthens the workloads fluctuations. Multi-level data layouts inevitably cause read amplification problems, such as those found at the LSM-Tree L0 level requires traversing multiple tables, so reading a single key-value pair may generate multiple I/Os. Third, the speed limit mechanism of cloud storage volume conflicts with LSM stores' internal multi-thread concurrency mechanism, and requests among multiple threads congestion on the cloud storage volume leads to an increase in latency multiples. Fourth, LSM stores' internally inherent mechanisms amplify the damage on the read latency of cloud storage volumes. Irregular flush operations or indeterminate size compaction operations cause a sudden increase in the number of I/Os accessing the cloud storage volume, resulting in high tail latency. Finally, the tradeoff between cost and performance increases the cost exponentially to get better tail latency, resulting in significant resource waste and limited throughput improvement.

One natural solution to the above challenges is contracting a higher IOPS budget with cloud storage volumes, ensuring that the LSM store's I/O number do not exceed the paid IOPS to maintain optimal latency. However, this raises the costs. Also, the peak IOPS demand in real production environments is difficult to predict. Instead, we aim to explore the best performance of an LSM store under a specific IOPS budget.

This paper presents Calcspar, a cloud storage volume contract-aware LSM store based on Amazon's EBS with reduced latency spikes, and it tolerates both external workload fluctuations and internal operation contentions. Calcspar first employs fluctuation-aware caching, which combines hotspot-aware proactive prefetching and shift-aware passive caching, to adapt to changing workloads. The prefetching strategy identifies hotspots for high load periods and proactively fetches them during low load periods, thus smoothing out the external load changes. Then, during the high load periods, the passive caching leverages the temporal locality to extrude the stale prefetched data and adapt to hotspot shifts without issuing extra requests. Calcspar then leverages a congestion-aware IOPS allocator to assign priority for different internal requests and avoid elevated latency due to limited IOPS budgets. The allocator employs a multi-queues throttling structure to prevent thread congestion. The opportunistic compaction then assigns write requests in different LSM levels into different priority queues, thus balancing the read amplification and write throttling. The contributions of this paper include:

1) We conducted an in-depth analysis of the performance of cloud storage volumes, which first illustrates the unwritten contract between latency and load pressures.

2) We propose a rate-limiting performance model for cloud storage volumes based on the observations, experimentally validate the model and reveal opportunities to obtain optimal latency.

3) Our proposed Calcspar is better suitable for AWS cloud storage volumes where IOPS budgets are vital to the performance and significantly reduced the tail latency of LSM-Tree.

The rest of this paper is organized as follows. Section 2 takes Amazon's EBS as the example to models the performance characteristics of cloud storage volumes. The challenges of reducing the latency for an LSM store on cloud storage are discussed in section 3. Calcspar designs are then introduced in section 4 to address these challenges and they are evaluated in section 5. Finally, the related work and conclusions are presented in sections 7 and 8, respectively.

2 Modeling Cloud Storage Performance

2.1 Contract Model of Cloud Storage

The cloud storage providers, such as AWS, offer a variety of cost-efficient storage volumes for users to meet their distinct needs and adapt to the changing market. Table 1 shows the contract model of the cloud storage, which illustrates the price and performance relationship of the corresponding volume type. The pricing is based on block storage in the AWS ap-northeast-2 region in July 2022 [3, 4]. The contract model indicates that as the price of IOPS increases, the lower latency of the corresponding type. Thus, it entails users choosing the appropriate storage volume and IOPS budget based on their needs. However, the paid IOPS only guarantees the number of returned I/Os rather than the optimal latency. Also, EBS performance scaling supports only increasing paid IOPS and takes hours to days to take effect [2]. There's no agreement on how the request would be responded to when the loads exceed the IOPS. Hence the corresponding latency characteristics are widely ignored by existing LSM stores.

2.2 Unwritten Latency Performance

To unwrap the hidden latency characteristics and understand how the above contract model would affect the performance

Table 1: EBS IOPS prices and latencies.

Type	Init IOPS	IOPS price ($)	Latency (μs)
gp2	3×GB	0.038	∼200
gp3	3000	0.0058	∼300
io1	100	0.0666	∼100
io2	100	0.067	∼10

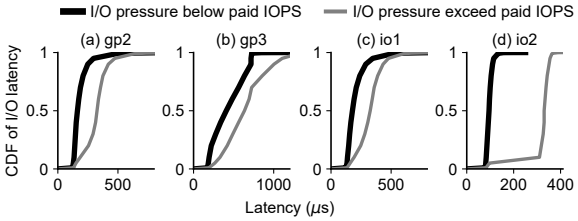


Figure 1: Latency CDF of different cloud storage.

of an LSM store, we first perform a series of experiments on cloud storage volumes, then proposing a performance model.

Experiment #1: *Cumulative Distribution Function (CDF) of latency under vary I/O pressures.* We measure the latency of EBS volumes gp2, gp3, io1, and io2 with paid 3000 IOPS for each by sending 4KB random read requests with varying pressures. We employ fio [22] to tune the I/O pressure by controlling the size of **Submit IOPS**, which is the number of I/Os submitted to EBS per second. Yet, the cloud storage volume *won't* handle more than the paid IOPS. Figure 1 shows the latency CDF results that support the following two findings.

Finding 1: When the I/O pressure exceeds the paid IOPS, the latency increases deterministically and significantly. On the contrary, their average latency performance is much better when the Submit IOPS is under paid IOPS. For example, the average latency of io2 even reached 100μs.

Finding 2: The IOPS budget is proportional to the cost when considering Table 1. The slightly higher-cost io2 has the best and most stable latency. The latency performance of the cheapest gp3, which initially provides 3000 IOPS, is far lower than the other three EBS types.

Experiment #2: *Limited IOPS budgets*. In this experiment, we explore the latency CDF under different IOPS budgets. We evaluate one io2 under two different pressures. Request latency distributions are given in Figure 2. Results indicate that regardless of the paid IOPS, the access latency when the Submit IOPS exceeds the paid IOPS is more than 5×

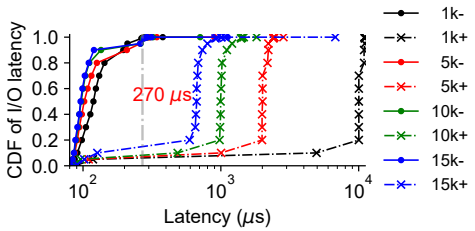


Figure 2: Latency CDF under different paid IOPS. "1k" means io2's paid IOPS. "+" indicates that the Submit IOPS exceeds the paid IOPS; "-" means not exceed.

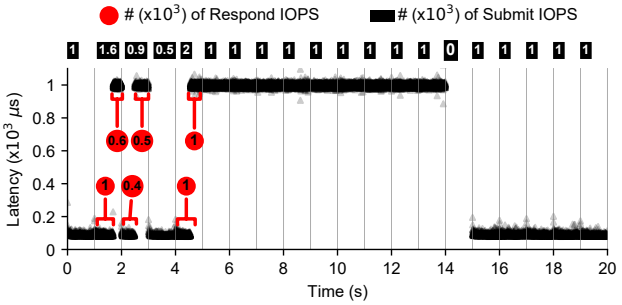


Figure 3: Latencies under rigorously controlled pressure. Respond IOPS is the number of requests returned.

worse than the access latency when not exceeded. The 99th percentile latency are below 270μs when the Submit IOPS does not exceed the corresponding paid IOPS. However, when the Submit IOPS exceeds the paid IOPS, the access latency increases to 1000∼11000μs. These long-tail latencies degrade the user experience.

Experiment #3: *Elevated latency spike*. To explore reasons behind the latency spike under high I/O pressures, we rigorously control the I/O send rate in a single thread for an 1000-IOPS io2 volumes. The latency of each request is shown in Figure 3. In the 1st second, when the Submit IOPS does not exceed the paid IOPS, the latency is lower than 200μs. In 2nd second, the Submit IOPS is 1600, the latencies of the first 1000 requests are identical to that of the first second. However, the latencies of the rest 600 requests increase significantly to about 1000μs, which renders $1/IOPS$ second. The Submit IOPS in the 5th second is twice the paid IOPS, the first 1000 requests can get low latency while the latencies of the last 1000 requests equal $1/IOPS$ second again. Although the Submit IOPS drops to 1000, the latencies of subsequent requests remain high. Until we pause the workload at the 14th second and resume it at the 15th second, the latencies recuperate.

Speculative Reason #1: *We speculate the reason behind the observation is due to the speed-limiting mechanism inside EBS, which handles the current excess I/O by overdrawing the next 1 second of IOPS, and at the same time, the "punitive" improvement latency is 1/IOPS to prevent the requests beyond the payment from continuing to be responded.*

For example, the last 600 I/O requests in the 2nd-second overdraw 600 IOPS from the 3rd second. Hence, only the remaining 400 (=1000 − 600) can be served quickly in the 3rd second. The overdraft is paid off when no I/O request is sent in the 14th second. Therefore, the latency returns to a lower level in the following 15∼19 seconds

Since the resources of cloud services are on a pay-per-use basis, cloud storage providers use this mechanism to maintain SLAs to prevent users from constantly acquiring benefits beyond what they paid. Meanwhile, by increasing the delay, the operation continues from the user's perspective; thus, there is no opportunity for recalling the service.

Experiment #4: *Thread congestion.* The effect of the num-

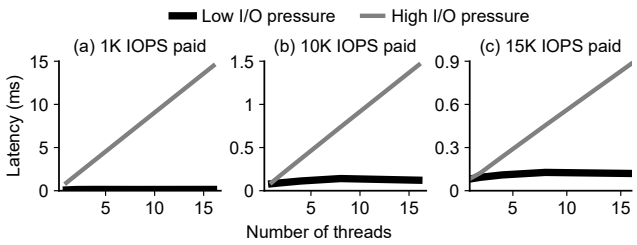


Figure 4: Average latency under different # of threads.

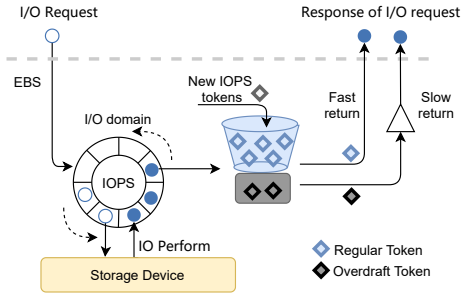


Figure 5: Estimated EBS IOPS throttling mechanism.

ber of threads on the latency is evaluated in this experiment. Two different I/O pressures are sent to each io2 volume with different number of threads. Request average latencies are shown in Figure 4. When the Submit IOPS does not exceed the paid IOPS, the average latency holds around 120μs and does not increase with more threads. However, when the Submit IOPS is higher than the IOPS paid, the access latency first grows to the level of $1/IOPS$ seconds while growing linearly with the number of threads. This phenomenon is consistent with queuing theory [30], so we conjecture the following based on the knowledge gained from queuing theory:

Speculative Reason #2: *Requests are queued when they enter the EBS, and the queue size equals the paid IOPS. The latency in the EBS will follow the following two rules: 1) When the Submit IOPS is lower than the paid IOPS, the latency of each request will be around the threshold, depending on the EBS type. 2) When the Submit IOPS exceeds the paid IOPS, the latency (respone time W) conforms to the queueing equation $W = L/A$, where A is the paid IOPS and L is the number of threads.*

2.3 Latency Model of EBS

Based on the above analysis, we construct an EBS latency model to reveal the internal latency mechanism of cloud storage, as shown in Figure 5.

Suppose an EBS volume a buffer queue (called I/O domain) to maintain requests, where the queue length is equal to the paid IOPS. When a request arrives, the I/O domain allocates a free slot in the queue. EBS then pulls requests from the I/O domain to promptly executes them. When the number of the responded requests exceeds paid IOPS, the EBS stops fetching new requests. Consequently, the pending requests are blocked until EBS can resume the services. Congestion can occur under multi-threaded workloads, although the total number of requests submitted by all threads is estimated to be no higher than the budget in one second. This is because thread scheduling uncertainties would fire some working threads more often, thus accidentally overdrawing the budget. Such an overdrawing leads the consequent requests to be blocked, resulting in significantly higher latency per request.

Overdraft Rule. EBS controls the responding speed of the request to manage the *rotation* of the I/O domain. Specifically, we use the “tokens" to describe the speed control mechanism of EBS. Each cloud storage volume retains a token bucket and a borrowing pool, which contains the same number of tokens equal to the paid IOPS. The user request first gets the regular token from the token bucket. If no token is in the bucket, the overdraft token must be obtained from the borrowing pool. EBS ensures that requests carrying regular tokens return quickly. In contrast, requests with overdraft tokens are processed slowly to ensure that the user does not use too many IOPS to maintain SLAs. The EBS is replenished with IOPS tokens per second, which are prioritized in the borrowing pool.

The above EBS latency model explains the aforementioned findings (#1 and #2) and speculations (reasons #1 and #2) about cloud storage latency: 1) When the Submit IOPS does not exceed the paid IOPS, requests get regular tokens and return quickly, and requests between threads do not block, so the optimal latency is obtained. 2) When the Submit IOPS exceeds the paid IOPS, some requests obtain overdraft tokens. 3) When the return speed is lower than the request arrival speed, the unprocessed requests will fill the I/O domain slots, and further tokens are replenished by replenishing the borrowing pool first so that the high latency state will last for a long time. 4) The inter-thread blocking in I/O domain leads to increasing user-perceived latency.

3 Modeling RocksDB Performance

3.1 RocksDB under IOPS Limitation

As suggested by the above performance model, cloud storage emphasizes more limitations on latency and IOPS rather than bandwidth. The evidence is that the allowed request size is relatively large, and increasing the IOPS budget is expensive. Such a contract and performance model fits bandwidth-sensitive workloads; however, it is unfavored by many latency-sensitive and request-heavy workloads. For example, our analysis indicates that the data write and compaction in RocksDB works well on cloud storage as its’ request sizes are more significant and the number of requests is limited. However, the read performance of RocksDB is seriously affected by the limited IOPS budget. Many applications that employ RocksDB to serve metadata indexing expect excellent read throughput as well as low and stable read latency [27]. Unfortunately, they will fail to achieve their purpose if the underline device

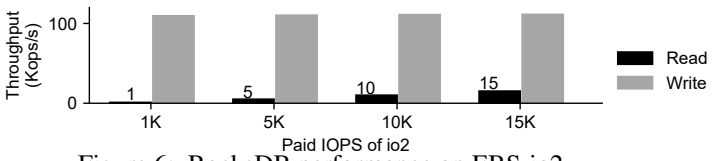


Figure 6: RocksDB performance on EBS-io2.

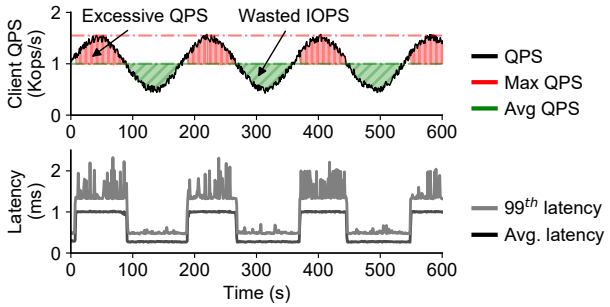


Figure 7: Client random read QPS and request latency with single thread.

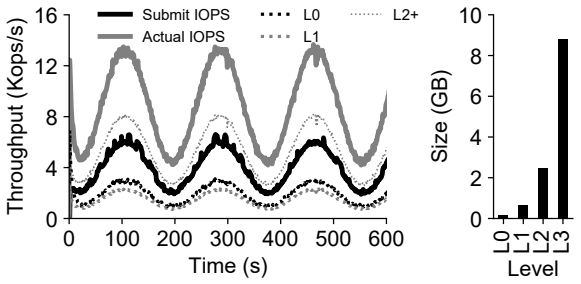


Figure 8: Read amplification of RocksDB.

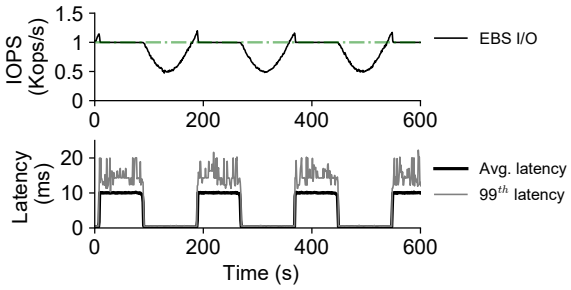


Figure 9: Io2 Respond IOPS and latency with 10 threads.

is cloud storage.

RocksDB does not work well on cloud storage because its LSM-Tree is generally a write-optimized indexing structure instead of tuned for reducing the number of read I/Os. Moreover, the hierarchical structure of the LSM-Tree results in many read requests needing to access multiple levels of the indexing table to locate the corresponding key. On the contrary, write operations can be cached and aggregated into large chunks before they finally hit the cloud storage in one backend request. As shown in Figure 6, we perform read and write stress tests on io2 storage volumes with different paid IOPS. As the IOPS increases, the read throughput increases, and the write throughput remains the same. Therefore, when deploying RocksDB on the cloud, the maximum 1000MB/s bandwidth of the cloud storage volume is usually sufficient. However, the RocksDB read I/Os will easily and repeatedly hit the IOPS limitations, which causes elevated tail latency.

3.2 Challenges in Avoiding Latency Spikes

This subsection elaborates on several challenges encountered when optimizing the read performance of RocksDB in cloud storage. To analyze the performance, we employ Facebook's most recent benchmark Mixgraph [12], which synthetically generates key-value requests that accurately represent the real-platform load fluctuation.

Challenge #1: *The read latency fluctuates significantly because cloud storage isn't flexible enough to meet the changing demand.* In this experiment, we send fluctuating read requests to an io2 volume in a single thread. To reduce expense, the paid IOPS of io2 storage volumes is the average of fluctuating requests. The experimental results in Figure 7 show that RocksDB is significantly affected by the latency characteristics of cloud storage volumes. When the client queries per second (QPS) exceeds the paid IOPS, the QPS being executed (the white part under the black line) can exceed the paid IOPS in a brief period, which is due to the overdraft rule. Then executed QPS will fall to the paid IOPS and the user-perceived latency is significantly higher than during low load. This also results in the user's excessive requests not being executed (the red part of the top in Figure 7), and they will be blocked. As the number of client requests drops below the paid IOPS, the latency drops to the bottom level, which causes another problem where the paid IOPS are wasted during low-load periods, as shown in green in Figure 7.

Challenge #2: *The read amplification in LSM-Tree further strengthens the workloads fluctuation.* RocksDB's write aggregation and hierarchical data layouts result in significant read I/O amplification. As shown in Figure 8, the actual IOPS is about 3× of the submit IOPS, amplifying the volatility of the load, when the access I/O exceeds the paid IOPS, the request latency will be high. In addition, the L0 and L1 level, which occupy a very small amount of data, but taking up close to 1/3 of the I/O accesses.

Challenge #3: *Thread I/O competition.* Requests among multiple threads are congested in the I/O domain resulting in a multiplication of tail latency. We send the same QPS as in Figure 7 with ten threads. The results in Figure 9 show that the latency increases 10× at high loads compared to single threads. This is because with more requests being sent to the io2 per second, the requests are returned slower than the requests that enter the I/O domain. When the I/O domain is full, requests are queued on each thread, so the latency perception is a multiple of the thread.

Challenge #4: *Bulk write blocking.* We sent some write requests while keeping the read QPS constant to measure the impact of write operations on user read requests, and the results are shown in Figure 10. At the 70th second, RocksDB launched compaction operations, which took up more IOPS,

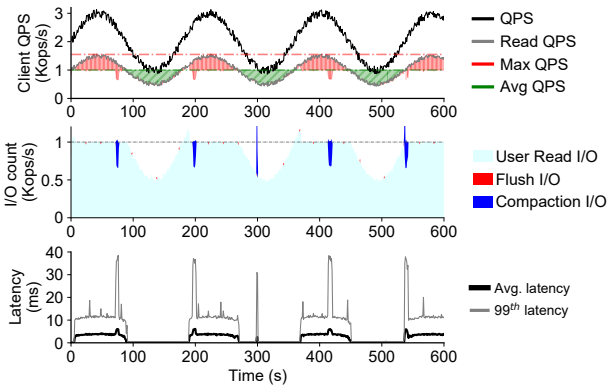


Figure 10: RocksDB I/Os and latency for a mixed read and write load with 10 threads.

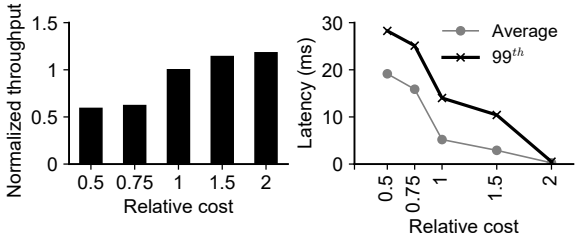


Figure 11: RocksDB performance at different storage costs.

causing some of the user's requests to wait in a queue. Hence the 99th percentile latency bursts to 40ms. Secondly, at the 300th second, even in low workload period, RocksDB internally initiates some compression operations, which blocks user requests, resulting in high latency.

Challenge #5: *Performance vs. cost.* We choose the average of the highest QPS and the lowest QPS of the load as the benchmark cost to measure the performance of RocksDB when choosing storage volumes with different costs under the same load. Results in Figure 11 show that, as the cost increases, although the latency appears to decrease, the throughput does not improve due to serious resource waste.

4 Calcspar Design

To build an LSM store that fully exploit the optimal latency of cloud storage volumes, the high latency caused by the observed overdraft rules and thread I/O congestion must be addressed. Rather than simply increasing expenses to improve the paid IOPS of storage volumes to avoid overdraft and congestion, we propose Calcspar to investigate the optimal latency of LSM stores in a cost-effective manner. With a limited number of IOPS available per second, Calcspar's four designs in Figure 12 holistically answer questions on getting the optimal latency: (1) How to smooth out I/O plateaus that are higher than the paid IOPS (§4.1 and §4.3). (2) How to take the most of available I/Os per second for user and LSM internal I/O requests (§4.2 and §4.4).

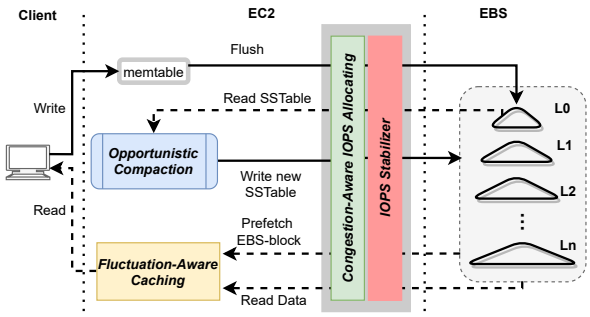


Figure 12: Architecture of Calcspar.

4.1 IOPS Stabilizer for EBS

Calcspar first aims to prevent latency fluctuation from the EBS. As discussed in §2.2, the overdraft rules do not result in throughput improvements but higher processing latency for EBS under high request pressures. Once the EBS enters the overdraft status, the application is not able to withdraw any pending requests, thus missing opportunities for further optimization but waiting. Rather than passively detecting unexpected latency spikes, Calcspar proactively controls the number of I/Os during high-load periods by only submitting requests with the highest priority. To achieve this, Calcspar employs the observed EBS latency model (§2.3).

Figure 13 details the IOPS Stabilizer, which throttles the request rate to match the EBS I/O budget, thus eliminating overdraft latency spikes. The essence is to mimic the token speed limit mechanism, which insides EBS and is widely ignored, to the upper-layer applications. We demand each request must obtain a token before accessing the EBS. The number of tokens is refreshed every second decided by the paid IOPS. By controlling the number of tokens, Calcspar guarantees requests sent to EBS do not exceed the paid IOPS, thus EBS processing latencies can be secured in the tens of microseconds. Consequently, applications can expect more stable latencies once a request successfully obtains a token.

4.2 Congestion-Aware IOPS Allocating

The second goal of Calcspar is to eliminate the latency spike caused by request congestion among threads. To minimize cloud storage costs, we assume the paid IOPS is only guaranteed to meet I/Os for the average usage. Hence, the muti-threads design adopted by modern LSM stores will inevitably congest due to limited tokens provided by the IOPS Stabilizer. Many works prioritize the execution of latency-sensitive I/Os by adjusting the I/O stack or leveraging multi-device parallelism [11, 20, 26]. However, they can not prevent less critical I/O requests from occupying precious available tokens in each time period (e.g. one second). To solve this problem, Calcspar uses multi queues of different priority together with the time window policy to ensure that critical requests are not occasionally blocked and are served in a best-effort manner.

Multi-priority Queues. Calcspar employs a multi-priority

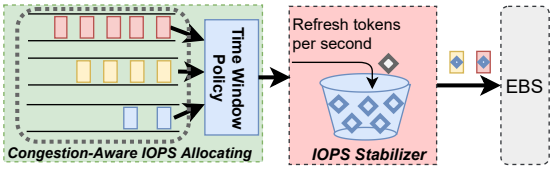


Figure 13: Congestion-Aware IOPS Allocator and Stabilizer

queue mechanism to categorize different I/O requests and allocate I/O tokens based on a dynamic time window policy. Calcspar treats I/Os that straightly affecting the LSM store as *user-aware requests*, and classifies I/Os that do not instantly hinder user requests as *non-user-aware requests*. User-aware requests are mainly for responding foreground user requests, so Calcspar places them into the highest priority queue. Non-user-aware requests are mainly from LSM background tasks (e.g., compactions and prefetching). Since different background tasks impact the read/write amplifications differently, Calcspar assigns them to medium-priority or low-priority queues. For example, prefetching requests go into the lowest priority queue. Note that compaction requests are further differentiated later (§4.4).

Dynamic Time Window Policy. Calcspar adopts a dynamic time window policy to optimize the utilization of paid I/Os per second. Under this policy, a time window represents a period within a second during which Calcspar grants requests to acquire tokens on a best-effort basis, so the size of a window matches the priority of the queue. The time windows of the queues are aligned at the tail within each one-second period, enabling Calcspar to prioritize processing I/Os from higher priority queues before those from lower priority queues. This ensures that requests from the highest priority queue are processed first, minimizing the chances of them being blocked by middle- or low-priority requests. Specifically, Calcspar always allocates a one-second time window for the high-priority queue. For other queues, Calcspar dynamically adjusts their time window sizes based on the allocated tokens in the previous second, using the formula `Allocated_IOPS` / `Paid_IOPS`. For example, during a one-second period, Calcspar assigns the time windows `[0,1)`, `[0.7,1)`, and `[0.9,1)` to the high-, middle-, and low-priority queues, respectively. In this case, the time window `[0.9,1)` signifies that requests in the low-priority queue cannot acquire tokens until the 0.9th second. Conversely, requests in the high-priority queue with the time window `[0,1)` are eligible to compete for tokens upon arrival.

4.3 Fluctuation-Aware Caching

Considering the read I/O amplification problem of an LSM store will cause significant latency spikes on the EBS, an EBS latency-aware cache plays an essential role on flattening I/O request plateaus to the EBS when the workload is heavy as well as improving the paid IOPS utilization when the workload is light. We find few existing cache schemes are designed on this purpose, and their design metrics do not take into account the available paid IOPS of the underlying EBS. This will significantly affect choices of the optimal cache policy when workload fluctuates.

Hotspot-Aware Proactive Prefetching. When the workload is light, Calcspar consumes spare paid IOPS to trade for a better cache hit ratio by prefetching SSTable. Calcspar manage data in the unit of EBS-block (e.g., 256KB, which is the maximum size allowed by the EBS for one I/O request), this ensures that each prefetching I/O reads as many data as possible. Calcspar then maintains a global table to track the hotness of EBS-blocks using the exponential smoothing algorithm based on their access history. Furthermore, Calcspar periodically and proactively rewarms the frequently accessed LSM top layer (e.g., L0 and L1) data, because LSM stores retrieves key-value pairs in a top-to-bottom layer fashion.

Shift-Aware Passive Caching. When the workload is heavy, an EBS latency-aware cache should minimizes its I/Os to the EBS while improving space efficiency. Calcspar manages the cache space passively in this case. Calcspar refines cache space management in the unit of 4KB and uses the LRU policy for better space efficiency because evicting any data can be punished by competing one I/O with user requests to access the EBS.

Cache Integration. Calcspar integrates the two cache policies introduced above and switches between them based on workload. These two policies manage the same cache space, but at any time, only one is active and evicts data. When the highest priority queue requests consume more than 95% of the tokens, Calcspar considers the workload to be heavy and activates the Shift-Aware Passive Caching policy. Otherwise, Calcspar harvests the available paid IOPS using the Hotspot-Aware Proactive Prefetching policy. It is worth noting that the global track table has a negligible memory overhead, as 1GB of data requires about 64KB of memory. Furthermore, the minimal cache pollution resulting from coarse-grained hotspot-aware proactive prefetching during periods of low load does not lead to a degradation in overall performance. This is because only frequently accessed data, known as hot data, is loaded based on historical access patterns. Additionally, given the ample IOPS budget available during low load and the consistently low latency of cloud storage, the penalty incurred from cache misses is negligible.

4.4 Opportunistic Compaction

The last goal of Calcspar is to remedy LSM compaction I/Os. After launching a compaction, its bulk read operations on at least two SSTables and write operations on at least one SSTable will compete with user I/O requests. An LSM store, on the other hand, retrieves a key-value pair level by level and merges SSTables in a copy-on-write manner, providing Calcspar with opportunities for differentiating I/Os for compaction jobs on different levels, thereby mitigating the competition on paid IOPS from LSM compaction operations. For L0 SSTables, which significantly affects read I/O amplifications,

Calcspar prioritizes compaction on them. For L1 and L2 SSTables, Calcspar puts their compaction I/Os into the medium priority queue, where they are opportunistically processed. As for SSTables in levels below L2, Calcspar assigns these compaction I/Os to the lowest priority queue, since short-term deferral has no noticeably affect on performance.

5 Evaluation

We implement Calcspar based on RocksDB and evaluate it to demonstrate its advantages. Specifically, we perform an extensive time delay to answer the following questions. (1) How does Calcspar perform compared to the state-of-the-art approach? (§5.2) (2) The impact of several techniques of Calcspar on performance. (§5.3, §5.4, §5.5) (3) The sensitivity analysis of Calcspar (§5.6).

5.1 Experimental Setup

Test platform. We employ the most widely deployed AWS as our test platform. The EC2 instance is m5d.2xlarge, configured with 8 vCPUs and 32 GB Memory. A representative io2 storage volume with 100 GB capacity and 1000 IOPS is used by default for performance evaluation.

Comparisons. We compare Calcspar with RocksDB and the other three state-of-the-art key-value stores. They are: 1) Autotuned RocksDB [6]: isolating the I/O bandwidth between user requests and internal flush/compaction operations to improve tail latency. 2) SILK [10]: Opportunistically allocates bandwidth to different internal I/O operations and allows low-level Compaction preemption. 3) CruiseDB [25]: Maintains SLAs employing an adaptive access mechanism based on memory usage, removes L0, and optimizes memory buffer. To make a fair comparison, all the databases take 4 threads for compaction and 4 for flushing. The default key, value, and SSTable sizes are set to 16B, 256B, and 8MB. The size of Memtables is set to the default value for RocksDB. A cache space of 500MB is opened for each database.

Table 2: YCSB workload characteristics.

Workload	Description
A	write-intensive:50% Update, 50% Read, Zipfian
B	read-intensive: 5% Update, 95% Read, Zipfian
C	read-only: 100% Read, Zipfian
D	read-latest: 95% Read, 5% Insert, Latest
E	scan-intensive: 5% Update, 95% Scan, Zipfian
F	write-intensive:50%Read,50%read-modify-write,Zipfian

Benchmarks. Two benchmarks, Mixgraph [12] and YCSB [15] are used to evaluate performance. YCSB is a widely used benchmark for evaluating the key-value store systems, providing six workloads configurations and key-value pair access distribution models listed in Table 2. YCSB can also provide uniform distribution workloads. Mixgraph is the latest benchmark test developed by Facebook. The workload is more spatially localized to simulate Facebook production workloads better and generate more accurate key-value queries. Benchmarks run in 10 threads in all the key-value stores by default, except for SILK, as multi-threading is not supported. In all experiments, 100 million Key-value pairs are first inserted into the key-value store system, and the key-value store has about 25 GB of data in its initial state.

5.2 Overall Performance

We first use the latest Mixgraph with fluctuating load characteristics to evaluate the overall performance of the five key-value stores on the cloud. Then we evaluate the performance using the YCSB benchmark and explore the effect under uniform load using the YCSB benchmark. To guarantee the fairness of the evaluation, the hardware resource allocation is the same for each key-value store. All evaluations start by randomly writing 100 million key-value pairs and executing one million requests.

The Mixgraph benchmarks. Figure 14 shows the performance for different read/write ratio configurations under Mixgraph. Since read requests affect IOPS more, the ratio of read requests is increased by varying the number of write requests. Based on the comparison of test results, the following conclusions can be drawn: 1) The throughput of Calcspar is better than other systems under all read ratios. In Figure 14(a), Calcspar throughput exceeds paid IOPS the most because of the high spatial locality of the Mixgraph load that is fully exploited. 2) Calcspar significantly reduces the average latency. As seen in Figure 14(b), the average latency of Calcspar does not exceed 200μs, which is 45$\sim$66% lower than the average latency of other key-value store systems. 3) Calcspar achieves a lower and more stable tail latency. Figure 14(c) shows the statistical plot of 99th percentile latency, and it can be seen that Calcspar has the smallest box plot volume with almost no outliers of extra-long delays. The 99th percentile latency can be stabilized at around 0.55ms with minimal fluctuations. CruiseDB also reduces tail latency by limiting request access, but it is unstable and sacrifices throughput.

The YCSB benchmarks. We use YCSB workloads with stronger time locality for performance evaluation. Figure 15 and Figure 16 show the throughput and the 99th percentile latency for each key-value stores system under the six workloads of YCSB. Calcspar can guarantee that the throughput is not lower than RocksDB in all cases, and the throughput can be improved under workload F. Throughput is not further improved due to the cloud storage IOPS budget constraints. Furthermore, the hot key of Zipfian distribution is scattered throughout the whole key space, resulting in underutilization of aggressive prefetching cache performance. Calcspar's main goal is optimizing latency, and as shown in Figure 16, the 99th percentile latency of Calcspar is reduced by 50% compared to other schemes. The best access latency is obtained because Calcspar is throttling the number of I/Os per second to access the cloud storage volume within a range of no more than paid IOPS. And cache redirection solves the queuing latency problem, so the latency per request is low. In other systems, requests beyond the paid IOPS will only to debit without strict

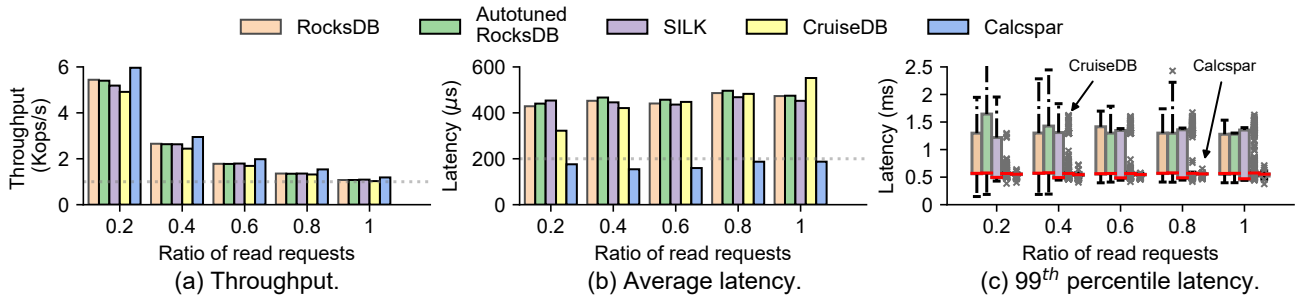


Figure 14: Evaluation results with different read request ratios under Mixgraph workload.

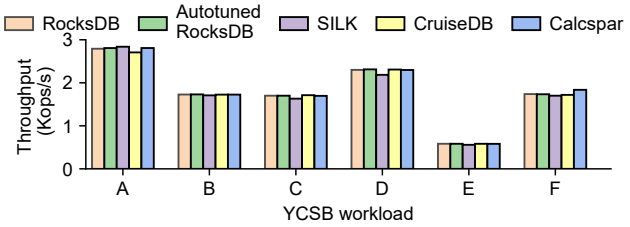


Figure 15: Throughput under YCSB workload.

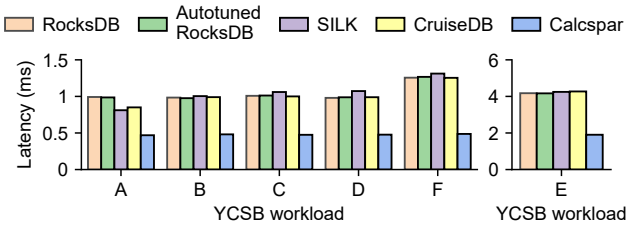


Figure 16: The 99th percentile latency under YCSB workload.

limits, and the request latency will be significantly higher.

For uniform workloads. We further use YCSB to evaluate the performance of calcspar under a uniform workload. Figure 17 shows that although the fluctuation-aware caching is less efficient under Uniform load, Calcspar exhibits shorter latency due to its flexible I/O throttling. CruiseDB's adaptive access mechanism can also reduce the average latency, but its tail latency increases to 20ms.

5.3 Congestion Mitigation Effectiveness

Here, we investigate the effect of Calcspar on solving thread congestion. We first test the latency performance under different threads and then explore the effect of the time window allocation IOPS strategy.

Avoid multi-thread congestion. Figure 18 shows the average latency and 99th percentile tail latency of the experiment running the default Mixgraph load on io2 with 1k paid IOPS using different user threads. Calcspar can keep the average latency at 175μs, 99th percentile latency always around 500μs, and the other schemes keep increasing both the average latency and 99th percentile latency as the number of threads increases. On cloud drives, the 99th percentile latency growth reaches 60$\times$ under 20 threads. Because other Key-value stores limit bandwidth without restricting the granularity of I/O. Therefore, as the number of threads increases, congestion becomes increasingly severe. Calcspar utilizes the EBS rate-limiting model, and the Congestion-Aware IOPS allocating avoids requests queuing in the I/O domain queue.

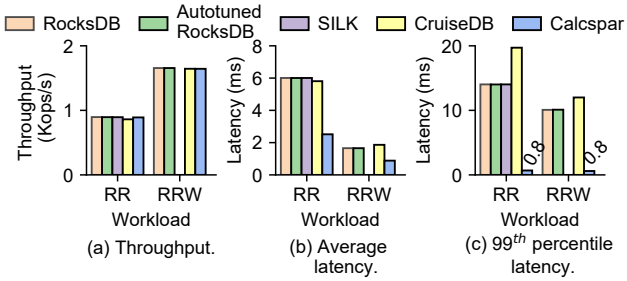


Figure 17: Evaluation results under Uniform workload. "RR" means random read, "RRW" means 50% random read and 50% random write.

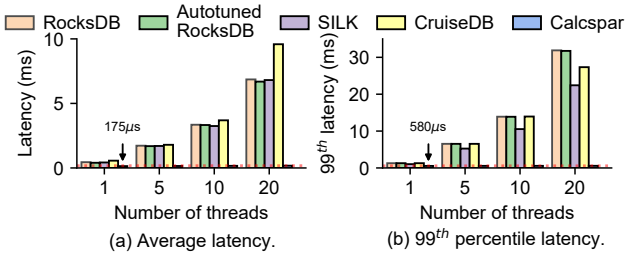


Figure 18: The latency with different user threads.

IOPS allocation strategy evaluation. We compared Calcspar's time window policy allocate IOPS (TWA) with three other IOPS allocation schemes: contention IOPS without allocation (NA), static allocation of IOPS among three queues in the ratio of 6:3:1 (SA), and dynamic allocation of IOPS based on the usage of the highest priority queue (DA). Using mixgraph load, where 5% are read requests and 95% are write operations, guarantees enough flush and compaction operations with 10 user threads running. The evaluation results in Figure 19 show that the time window strategy has a good throughput and 99th percentile latency is reduced by 50% compared to the NA. The SA has the worst performance because of resource wastage. The DA can fully utilize IOPS and the average latency is the lowest, but the 99th percentile latency is higher than TWA because the requests in other queues will block user requests.

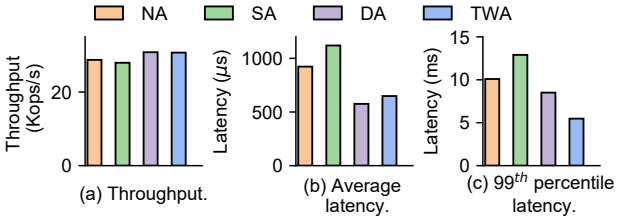


Figure 19: Performance of four IOPS allocation schemes.

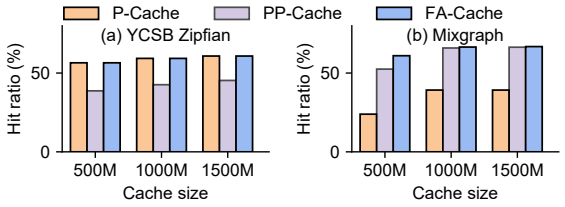


Figure 20: Hit ratios of different caching schemes under different workloads.

5.4 Cache Effectiveness

Then, we evaluate the effect of Fluctuation-Aware Caching regarding the cache hit ratio, the impact of cache size and the corresponding read amplification.

Cache Hit Ratio. We compared Calcspar's fluctuation-aware cache (FA-Cache) with only passive cache (P-Cache) and only proactive prefetching cache (PP-Cache) under YCSB and Mixgraph workloads. Figure 20 shows that FA-cache has the highest hit ratio under both workloads. For YCSB load, the hotkeys are randomly distributed in the key space, so it is more suitable for P-cache with small prefetching. However, under Mixgraph load, the hotkeys are relatively concentrated and more suitable for PP-cache. Calcspar's FA-cache combines these two advantages. Also, we can find less than 5% of data space can achieve up to 60% hit rate, which can increase the overall capacity of the system at a lower cost.

Impact of Cache Sizes. We increased the cache size from 0.1% to 20% of the total number to explore its impact on performance. Figure 21 shows that Calcspar outperforms RocksDB regarding latency at any cache size. With 1% cache size, Calcspar can reduce the average latency to below 200μs. RocksDB's block cache requires the cache size at least 5% of the total data to get an average latency close to 200μs, but the 99.9th percentile latency is still as high as 1200μs.

Read amplification. We count the number of read requests sent from the client side and the I/O accesses to the EBS to explore the effect of Calcspar in mitigating read amplification. We use Mixgraph default configuration for evaluation and

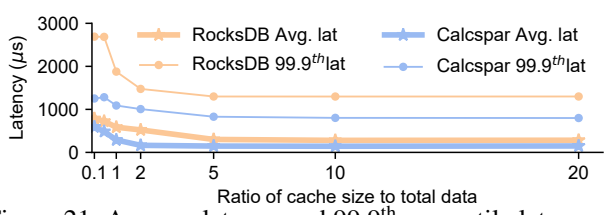


Figure 21: Average latency and 99.9th percentile latency of RocksDB and Calcspar with different cache sizes under Mixgraph load.

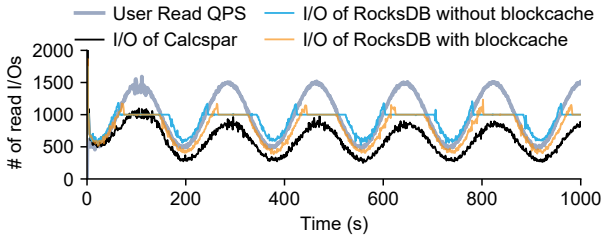


Figure 22: Read QPS and number of I/Os send to EBS.

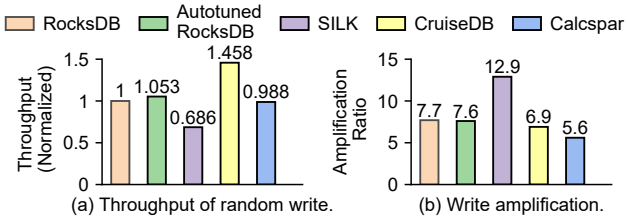


Figure 23: Write performance.

compare it with RocksDB without cache and RocksDB with the same size (500MB) blockcache turned on. In Figure 22, the results show that Calcspar sends the least number of I/O requests to EBS with the same user requests, even during peak periods, because the fluctuation-aware caching can cache L0 and L1 in advance during low-load periods. RocksDB with blockcache enabled is limited by the IOPS budget during high load, and RocksDB without blockcache enabled sends more I/O to EBS during low load because of read amplification.

5.5 Impact of Opportunistic Compaction

Write performance. We compare the performance of Calcspar with the rest of the schemes under full write load. Figure 23(a) shows the throughput of 10 threads writing 100 million key-value pairs randomly (except for SILK single threads). Compared to RocksDB, Calcspar's write performance is only 1.2% lower. Both Autotuned RocksDB and CruiseDB allocate bandwidth to prioritize upper-level write operations, which improves performance. SILK can only write in a single thread, so performance is poor.

Write amplification. Figure 23(b) shows that Calcspar reduces write amplification the most because Calcspar blocks L0 level to L1 compaction slightly. However, the write performance can not be improved because of IOPS allocation. CruiseDB removes the L0 level to reduce the write amplification. SILK prioritizes the execution of flush and L0 level compaction, resulting in frequent reads and writes of L1 level data, which in turn causes higher write amplification.

5.6 Sensitivity Analysis

Paid IOPS of EBS. We first evaluate five KV stores on io2 with different paid IOPS using Mixgraph to explore the impact of paid IOPS on performance, where Mixgraph uses the default read/write configuration and ensures that the average value of load fluctuations is equal to paid IOPS. Figure 24(a) shows that regardless of the paid IOPS of io2, Calcspar en-

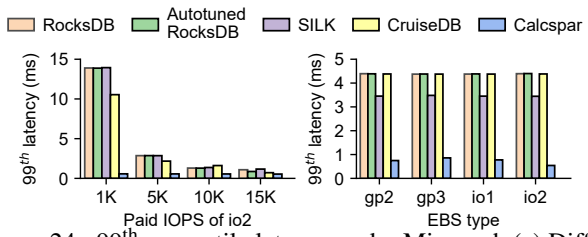


Figure 24: 99th percentile latency under Mixgraph.(a) Different paid iops. (b) Different EBS type.

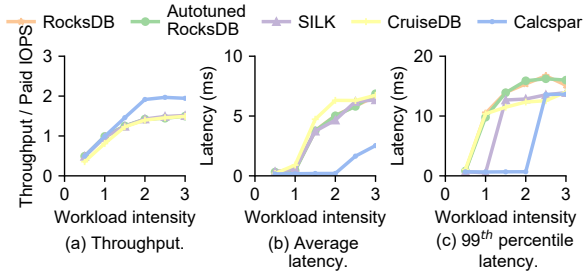


Figure 25: Performance at different workload intensities.

sures the 99th percentile latency stays low. As the paid IOPS increases, the request latency under high pressure becomes progressively smaller, but at 15k paid IOPS, Calcspar's 99th percentile latency is still 24% ~50% less than other solutions.

Type of EBS. We use four types of EBS volumes with 3000 IOPS to explore the applicability of Calcspar on AWS EBS. Figure 24(b) shows that Calcspar exhibits the lowest 99th percentile latency on all four EBS compared to the remaining four options. As the performance of the storage volume gets better, the 99th percentile latency of Calcspar gets lower, e.g., io2 has a lower 99th percentile latency than gp3 by 200µs. In contrast, the other schemes have a higher latency or no change. The results fully illustrate that our scheme is suitable for various types of cloud block storage devices in AWS.

Workload pressure. We evaluate the pressure resistance by varying the workload intensity, which is the ratio of the average read requests of Mixgraph fluctuating load to paid IOPS of io2. Figure 25(a) shows that Calcspar can handle up to twice the paid IOPS for read requests. Figure 25(b) shows that Calcspar can guarantee an average latency of 200μs even at twice the workload, while the rest of the solutions have higher latency when the read workload exceeds paid IOPS. The 99th percentile latency of Calcspar is still the smallest at high workloads in Figure 25(c). Overall, Calcspar has good pressure resistance and adaptability.

Different Cloud Vendors. To demonstrate the versatility of Calcspar, we conducted a performance comparison between Calcspar and vanilla RocksDB on three prominent cloud provider platforms, namely AWS, Alibaba Aliyun [7], and Microsoft Azure [29]. Our evaluation encompassed instances equipped with 8v CPUs and 32GB RAM, coupled with cloud block storage volumes configured with 5000 paid IOPS. Specifically, AWS utilized EBS io2, Aliyun utilized ESSD PL1, and Azure employed Azure managed disk Premium SSD v2. Figure 26 illustrates the evaluation results,

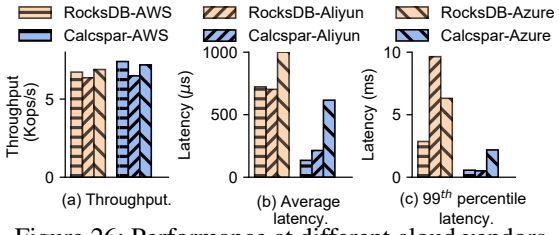


Figure 26: Performance at different cloud vendors.

highlighting Calcspar's better performance across all three cloud storage types when compared to vanilla RocksDB. On Aliyun, Calcspar exhibited notable achievements, with a significant decrease in the 99th percentile latency from 9.6ms to 509μs and an average latency reduction of 69.5%. These improvements arise from Aliyun's utilization of a throttling mechanism that triggers longer latencies when the number of requests exceeds the contractual agreement, which Calcspar is designed to avoid. While Azure managed disk Premium SSD v2 demonstrated comparatively higher latency than the other two cloud block storage devices, Calcspar still highlights its efficacy and managed to reduce both average and 99th percentile latencies.

6 Discussion

Cloud Storage Vendors. We selected AWS EBS as a representative cloud storage platform due to its significant market share (32%) and its provision of easy-to-use, high-performance block storage at any scale. While different cloud storage vendors may vary in terms of equipment and pricing for block storage volumes, they all follow an IOPS-based contract model to deliver services to the public. This model is also adopted by other providers such as Microsoft Azure [29], Google Cloud [18], and Alibaba Aliyun [7]. During periods of bursty large numbers of requests, it is common for paid IOPS to be exceeded, leading the cloud provider to enforce request processing limits in order to uphold the contract. However, simply restricting request processing or increasing the latency of each request would result in higher tail latency. Calcspar mitigates such issues through a series of techniques that can be applied across different cloud storage platforms. Figure 26 validates Calcspar's design on various representative cloud storage vendors.

Throttling Models. Cloud providers employ different measures to maintain contract compliance and minimize the impact on other users within the cloud environment. AWS EBS utilizes an overdraft rule, which offers little opportunity for applications to prioritize their request queues. Experiment 3 in Section 2.2 demonstrates that once requests are queued in EBS, they cannot be withdrawn to accommodate more urgent tasks. The overdraft rule, in effect, obscures the traffic congestion from users. On the other hand, blocks user requests for a relatively long duration, resulting in prolonged tail latency. This situation is unfavorable as it halts request processing during that period. Other cloud storage providers

employ varying throttling mechanisms, often achieved by introducing increased latency. Consequently, enabling priority masks or rejecting overdraft requests would be beneficial. Another possibility is to allow users to employ customized cache algorithms to digest temporary workload peaks.

7 Related Work

Latency-aware Storage Stack. Many works focus on providing fast I/O services by optimizing the I/O stack [11] to exploit the μs-scale latency of storage devices. Blk-switch [20] brings network switch techniques into the block storage stack and proposes an I/O scheduler, thus solving the head-of-line blocking problem and achieving low tail latency. FastResponse [26] targets on *ultra-low latency* SSDs, and it coordinates the scheduling of different I/O levels to mitigate I/O interferences. PAIO [28] proposes an I/O optimization framework, enabling flexible I/O scheduling policies through I/O information propagation. However, these efforts focus on either how to fully explore the potential of multi-core/hardware resources or how to alleviate contentions between latency-sensitive and throughput-demanding applications. Calcspar targets LSM-Tree key-value stores over the cloud storage, and Calcspar addresses the I/O contentions between user read I/Os and LSM internal I/Os, achieving low latency.

LSM Store Compaction. Compaction I/O operations within LSM-Tree will compete with user read operations, resulting in long-tail latency. There are approaches to reduce data writes by delaying or merging some compaction actions [31, 33, 41]. Some studies reduce contention for storage devices by tuning and scheduling internal tasks [9, 10, 13]. Some adaptive compaction schemes have also been proposed for performance optimizing [16, 35]. However, these are all compaction optimizations targeting bandwidth-constrained SSDs. Optimizing compaction operation alone is not enough, as user-read requests already dominate the paid IOPS.

Software and Hardware Co-design for Latency. Hardware and software coordination can better reduce long-tail latency. For example, RStore [24] fully utilizes the advantages of multiple cores to reduce the tail latency of in-memory key-value stores. BCW [37] achieves low write latency on HDDs by reshaping patterns that utilize the HDD internal buffer. Vigil-KV [23] demonstrates the latency state of NVMe SSDs using a predictable latency mode interface and ensures controllable tail latency by scheduling compaction/flush operations and client requests.

Data Cache. Prefetching or caching frequently accessed data to a high-performance cache device can greatly improve read performance by reducing the number of slow I/O operations. Leaper [40] leverages machine learning methods to predict hot data and proactively prefetch them to the cache. AC-Key [38] aims at LSM cache mechanisms in the memory, hybrids different kinds of cache objects, and dynamically adjusts their sizes to improve cache efficiency. To reduce cache invalidation due to hotspot shifting and internal compaction, A parallel cache prefetching method [43] is proposed to prefetch the most valuable blocks into the cache by hotspot key-value pair tracking. Thus, read operations are not affected by the compression. LSM-tree [36] uses a compaction buffer to minimize these cache pollutions.

Reduce Storage Cost. Cloud storage users are often sensitive to storage costs, and they usually hybrids cloud storage volumes of different prices to cut the overall storage cost. Mutant [42] controls the overall storage cost by adaptively tuning the size of expensive high-performance storage volumes. PrismDB [32] pines hot data in the upper LSM levels to reduce storage costs. RocksMash [39] stores all metadata and frequently accessed data in local storage, while putting the rest in the cloud for cost efficiency. SA-LSM [44] uses survival analysis algorithms to predict hot and cold data at records granularity, and schedules compaction with external services to reduce costs by storing hot and cold data separately. Calcspar trades higher IOPS capabilities with the smaller memory or higher performance storage devices as cache, rather than simply purchasing more IOPS for cloud volumes.

8 Conclusions

This paper profoundly explores and models the latency mechanism of AWS's EBS cloud storage. Experimental analyses show that limited IOPS budgets in EBS contracts cause high latency spikes to endpoint users when requests exceed a threshold. We specifically investigate the LSM-tree based key-value store, which both amplifies external workload fluctuations and develops internal request congestion. We propose Calcspar, a contract-aware LSM store for cloud storage with reduced latency spikes. The fluctuation-aware caching strategy in Calcspar reduces 99th percentile latency by more than 66% under varying workload pressures without incurring noticeable caching costs. The congestion-aware IOPS allocation further avoids up to 60$\times$ tail latency spikes by assigning different priorities for internal operations and preventing thread I/O contentions. Our sensitivity study demonstrates that Calcspar is generalizable for different cloud storage volumes. Accordingly, Calcspar can be offered as a companion service to cloud storage providers, significantly balancing the performance and cost for endpoint users.

Acknowledgments

We would like to thank our shepherd, Jana Giceva, and the anonymous reviewers for their valuable feedback and suggestion. This work was sponsored in part by the National Natural Science Foundation of China under Grant No.62072196 and No.62102155, the Key Research and Development Program of Guangdong Province under grant No.2021B0101400003, the Creative Research Group Project of NSFC No.61821003, the Fundamental Research Funds for the Central Universities (HUST: 2021XXJS034), and the Youth Foundation Project of Zhejiang Lab (No.K2023PI0AA04). Any views expressed are solely those of the authors and do not necessarily reflect the opinions or recommendations of the funding agencies.

References

[1] LevelDB. https://github.com/google/leveldb.

[2] Amazon EBS Volume Modify Limitations. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/modify-volume-requirements.html, 2022.

[3] Amazon EBS Volume Pricing. https://aws.amazon.com/cn/ebs/pricing, 2022.

[4] Amazon EBS Volume Types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html, 2022.

[5] RocksDB. https://github.com/facebook/rocksdb, 2022.

[6] RocksDB Autotuned Rate Limiter. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter, vistied Jan 2019.

[7] Alibaba Cloud.2023. Aliyun. https://www.alibabacloud.com/product/disk/, 2023.

[8] Amazon.2022. Cloud Storage. https://aws.amazon.com/what-is-cloud-storage/, 2022.

[9] 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 *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, 2017.

[10] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[11] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC 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.

[13] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, and Yangyang Wang. Adaptive lower-level driven compaction to optimize lsm-tree key-value stores. *IEEE Transactions on Knowledge and Data Engineering*, 34(6):2595–2609, 2022.

[14] 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. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[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 10)*, pages 143–154, 2010.

[16] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.

[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. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[18] Google.2023. Google Cloud. https://cloud.google.com/, 2023.

[19] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[20] 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.

[21] IDC, The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2022.

[22] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio, 2022.

[23] Miryeong Kwon, Seungjun Lee, Hyunkyu Choi, Jooyoung Hwang, and Myoungsoo Jung. Vigil-KV: Hardware-Software Co-Design to integrate strong latency determinism into Log-Structured merge Key-Value stores. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 755–772, Carlsbad, CA, July 2022. USENIX Association.

[24] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proc. VLDB Endow.*, 13(7):1091–1104, mar 2020.

[25] Junkai Liang and Yunpeng Chai. Cruisedb: An lsm-tree key-value store with both better tail throughput and tail latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1032–1043. IEEE, 2021.

[26] Mingzhe Liu, Haikun Liu, Chencheng Ye, Xiaofei Liao, Hai Jin, Yu Zhang, Ran Zheng, and Liting Hu. Towards Low-Latency I/O Services for Mixed Workloads Using Ultra-Low Latency SSDs. In *Proceedings of the 36th ACM International Conference on Supercomputing (ICS 22)*, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. Tridentkv: A read-optimized lsm-tree based KV store via adaptive indexing and space-efficient partitioning. *IEEE Trans. Parallel Distributed Syst.*, 33(8):1953–1966, 2022.

[28] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. PAIO: General, portable I/O optimizations with minor application modifications. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 413–428, Santa Clara, CA, February 2022. USENIX Association.

[29] Microsoft.2023. Azure. https://azure.microsoft.com/en-us/products/storage/disks/, 2023.

[30] Harchol-Balter Mor. *Performance modeling and design of computer systems*, volume 576. Cambridge University Press, 2013.

[31] Feng-Feng Pan, Yin-Liang Yue, and Jin Xiong. dcompaction: Speeding up compaction of the lsm-tree via delayed compaction. *Journal of Computer Science and Technology*, 32(1):41–54, 2017.

[32] Ashwini Raina, Asaf Cidon, Kyle Jamieson, and Michael J Freedman. Prismdb: Read-aware log-structured merge trees for heterogeneous storage. *arXiv e-prints*, pages arXiv–2008, 2020.

[33] 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*, pages 497–514, 2017.

[34] Reseach and Markets. Cloud Storage Market. https://www.researchandmarkets.com/reports/4306260/cloud-storagemarket-forecasts-from-2017-to-2022, 2022.

[35] Nicholas Joseph Ruta. *CuttleTree: Adaptive tuning for optimized log-structured merge trees*. PhD thesis, 2017.

[36] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. Lsbm-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 68–79. IEEE, 2017.

[37] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. BCW: Buffer-Controlled writes to HDDs for SSD-HDD hybrid storage server. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 253–266, Santa Clara, CA, February 2020. USENIX Association.

[38] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. AC-Key: Adaptive caching for LSM-based Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 603–615. USENIX Association, July 2020.

[39] Peng Xu, Nannan Zhao, Jiguang Wan, Wei Liu, Shuning Chen, Yuanhui Zhou, Hadeel Albahar, Hanyang Liu, Liu Tang, and Zhihu Tan. Building a fast and efficient lsm-tree store by integrating local storage with cloud storage. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(3):1–26, 2022.

[40] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proc. VLDB Endow.*, 13(12):1976–1989, jul 2020.

[41] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST)*, pages 1–13, 2017.

[42] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 18)*, pages 162–173, 2018.

[43] Shuo Zhang, Guangping Xu, YuLei Jia, Yanbing Xue, and Wenguang Zheng. Parallel cache prefetching for lsm-tree based store: From algorithm to evaluation. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 222–236. Springer, 2021.

[44] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. Sa-lsm: Optimize data layout for lsm-tree based storage using survival analysis. *Proc. VLDB Endow.*, 15(10):2161–2174, jun 2022.

Adaptive Online Cache Capacity Optimization via Lightweight Working Set Size Estimation at Scale

Rong Gu[1] Simian Li[1] Haipeng Dai[1] Hancheng Wang[1] Yili Luo[1] Bin Fan[2] Ran Ben Basat[3]
Ke Wang[4] Zhenyu Song[5] Shouwei Chen[2] Beinan Wang[2] Yihua Huang[1] Guihai Chen[1]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China[1]
Alluxio Inc[2] *University College London*[3] *Meta Inc*[4] *Princeton University*[5]

Abstract

Big data applications extensively use cache techniques to accelerate data access. A key challenge for improving cache utilization is provisioning a suitable cache size to fit the dynamic working set size (WSS) and understanding the related item repetition ratio (IRR) of the trace. We propose Cuki, an approximate data structure for efficiently estimating online WSS and IRR for variable-size item access with proven accuracy guarantee. Our solution is cache-friendly, thread-safe, and light-weighted in design. Based on that, we design an adaptive online cache capacity tuning mechanism. Moreover, Cuki can also be adapted to accurately estimate the cache miss ratio curve (MRC) online. We built Cuki as a lightweight plugin of the widely-used distributed file caching system Alluxio. Evaluation results show that Cuki has higher accuracy than four state-of-the-art algorithms by over an order of magnitude and with better stability in performance. The end-to-end data access experiments show that the adaptive cache tuning framework using Cuki reduces the table querying latency by 79% and improves the file reading throughput by 29% on average. Compared with the cutting-edge MRC approach, Cuki uses less memory and improves accuracy by around 73% on average. Cuki is deployed on one of the world's largest social platforms to run the Presto query workloads.

1 Introduction

Nowadays, distributed data-intensive frameworks like Flink [9], Spark [49], Presto [37], which frequently read data from tables and files, commonly use a caching layer as one key optimization to improve data accessing performance. However, allocating the right amount of cache storage can be non-trivial: excessive resource unnecessarily increases the cost, while insufficient capacity degrades the performance. Dynamic online workloads [24, 42] make this problem even more challenging. Particularly, when operating the Presto deployments, we introduced Alluxio [1, 25] as its caching layer and observed a high cache hit ratio. It was important, however, unclear to us based on existing cache metrics to tell if we could reduce the cache capacity of Presto servers while maintaining the high cache hit ratio.

Existing approaches about how to tune the cache capacity can be mainly summarized into four categories: (1) Rule-based approaches [21, 24, 34, 42] tune cache sizes based on cache metric related rules. However, it always adjusts the cache size to fit the working set size blindly and frequently. (2) ML-based approaches [4, 28, 30, 33, 35] train machine learning models with historical data offline and predict proper cache sizes in the future. Nevertheless, the model might be inaccurate under online dynamic workloads. (3) MRC-based approaches [15,19,22,36,40,41,45,51,52] explore the optimal cache size by exploring a miss ratio curve (MRC) as the function of the cache size. However, MRC is generated by assuming each item has the same size or cost, which is not always true in practice. (4) Window-based approaches [5, 11, 20] determine the cache capacity by estimating the cardinality of items in a sliding window. But, it can not estimate the working set size of items in variable size or organized in multiple scopes.

Understanding the online accurate working set size (WSS) and item repetition ratio (IRR) is important for tuning appropriate cache capacity [35, 51].

Accurate WSS estimation supports better cache capacity planning, leading to higher cache hit rate and significant end-to-end performance improvement. In the cluster, WSS and IRRs insight need to be captured in real-time since the data access load may vary dynamically. In addition, it is imperative to use low CPU and memory resource, as it is long-running and may scale to dozens and hundreds of nodes. What's more, to monitor WSS and IRR of various applications, it needs to track the online items that have different sizes and structure levels. To sum up, ideally, to effectively tune the cache size online, an accurate, time-efficient, dynamic, light-weighted approach for tracking the working set size (WSS) and variable-size item repetition ratio (IRR) in a sliding window is needed.

We propose Cuki, an approximate data structure for estimating the online WSS and IRR for variable-size item access with proven accuracy guarantee and little overhead in the slid-

ing window. Generally, we face three challenges in the design and usage of Cuki.

The first challenge is how to estimate the WSS and IRR online with little resource and proven accuracy. The item size can span over 8 ~ 9 orders of magnitude [24, 42] in the production environment. Therefore, inaccurate tracking items such as sampling may lose critical items, which may cause a huge drop in WSS estimation. The amount of data accessed in a time window can be quite large. It would be very time-inefficient and space-costly to store and calculate the item information online. To address this challenge, we carefully design a compact data structure with the approximate and item-wise tracking mechanisms.

The second challenge is how to achieve good scalability in high concurrency scenarios like multi-threading. It is common for real-world applications to access data concurrently. As the number of threads increases, the consistency and efficiency of concurrent access issues become obvious. To address this challenge, we adopt and propose a series of fine-grained concurrency control methods (§ 4), such as opportunistic aging.

The third challenge is how to judge the cache status under various scenarios with Cuki. It is non-trivial to tell whether the cache is overloaded or underused at a moment due to the variety of WSS and the cache-friendliness of the applications online. To address this challenge, we get the cache status insights by comparing the real cache size and the cache hit ratio with the WSS and IRR estimated by Cuki online, respectively.

By working with both Presto and Alluxio open source communities, our contributions can be summarized as follows:

- **Lightweight and Accurate WSS/IRR Estimation:** We design an approximate data structure, called Cuki, to estimate WSS and IRR online over a sliding window with little resource overhead and proven accuracy. Cuki uses an item-wise fine-grained tracking mechanism to reduce WSS error caused by missing critical items (*e.g.,* large ones). In our experiment, with a 96KB memory space size, Cuki can provide 99.07% accuracy for a 511MB working set size over the MSR data trace (§ 6.3). In addition, Cuki supports multi-scope WSS estimation with an easy feature extension.
- **Fine-grained Concurrency Control Methods:** To improve the efficiency of the concurrent access in Cuki, we propose opportunistic aging which decreases the lock contention risk in high concurrency scenarios. In addition, we adopt the segmented lock and two-phase based insertion mechanisms to guarantee data consistency in concurrent access.
- **Adaptive Online Cache Capacity Tuning Framework Using Cuki:** Finally, we propose an adaptive online cache capacity tuning mechanism based on Cuki. It judges whether the current workload, such as table querying and file reading, is cache-friendly or not, and further tells whether the cache system is overloaded or underused. Accordingly, the proposed cache capacity tuning mechanism can adjust the cache storage size online to fit current workloads.
- **Extensive Evaluation and Application Practice:** Experimental results on extensive benchmarks show that Cuki achieves over 10× higher accuracy with more stable performance compared with state-of-the-art methods. The cache tuning mechanism using Cuki can reduce the table reading latency by 79% on average, and improve the file reading throughput by 29% on average, respectively. Compared with the cutting-edge MRC approach, Cuki uses less memory and improves the accuracy by 73% on average. In addition, end-to-end real-world query workload experiments show that the proposed approach is effective for large-scale cache systems.

2 Background

Cuckoo Filter: A Cuckoo filter [17] is a well-known approximate data structure for deciding whether a given item is in a set or not. It consists of several buckets, and each bucket has four slots by default. Each item has two candidate buckets in a Cuckoo filter. To save space, a Cuckoo filter stores the fingerprint of an item rather than the item itself.

To insert item x, the Cuckoo filter first gets the fingerprint of x as f. Then, the Cuckoo filter hashes the x to get the first candidate bucket position b_1. The other candidate bucket position can be obtained by computing $b_2 = b_1 \oplus hash(f)$. The item x will be inserted into an empty slot of these two candidate buckets. If both candidate buckets are full, the Cuckoo filter relocates other items iteratively until it finds an empty slot. To check whether item x is already in the set, the Cuckoo filter first computes the two candidate buckets of x as described above. Then, it checks the items' fingerprints in these two buckets. If the Cuckoo filter finds one's fingerprint is the same as x's fingerprint, it returns true. Otherwise, it returns false.

Miss Ratio Curve: A key challenge of cache resource allocation is understanding the relationship between the cache hit ratio and the cache size. The miss ratio curve (MRC) is a common approach to figure out this relationship. The basic idea of MRC is to generate a miss ratio curve as a function of the cache size. With the generated miss ratio curves, users can allocate the cache size properly by observing the trend of the cache miss ratio with the cache size.

A traditional way [29] to generate a miss ratio curve of the specific trace is to compute the reuse distance of each item. The reuse distance of a specific item x represents how many items have been cached since the last access of the x. Since the reuse distance of each item has been recorded, this approach will give an ideal miss ratio curve. However, the online overhead of this approach is non-negligible.

To reduce the overhead of generating MRCs, recent research work use sampling techniques. Counter Stack [45] uses down-sampled and pruned probabilistic counters. SHARDS [41] samples the input trace. AET [22] uses average eviction time to construct MRC. Mini-sim [40] extends SHARDS by using miniature simulation.

However, these methods [22, 40, 41, 45] have three limitations. First, they need to store or process a separate I/O trace. Second, they use sampling techniques, which are likely to miss heavy hitters (large-sized items) and incur inaccuracy. Third, they focus on processing fixed-size item accesses and need some redesign to handle variable-size objects. RAR-CM [51] uses the hashmap to store the item access information and estimate the item repetition ratio (IRR) for generating an approximate MRC. However, RAR-CM is still primarily designed for fixed-size item access and needs 128 bits to store each item. Our Cuki only needs 52 bits for each item to support variable-size item access MRC generation and is around 73% more accurate than RAR-CM (§ 6.7).

Though the overhead for Cuki to generate MRC is low, MRC generation brings additional overhead for Cuki after all. Since WSS/IRR estimation is usually sufficient for cache size tuning in our environment, we finally choose the WSS/IRR estimation as the main approach.

Prior Cache Size Tuning Approaches: The critical difficulty in improving cache utilization is tuning the proper cache size online with limited resource. Prior approaches can be mainly summarized into four categories:

- **Rule-based:** Rule-based approaches [21, 24, 34, 42] observe cache usage metrics. If the observed metrics exceed or are less than the predefined threshold, it tunes the cache size. For example, Pocket [24] increases the cache size when cache usage exceeds 80%. However, due to lacking knowledge of the working set sizes of online workloads, it does not know what the best cache size should be tuned to each time.
- **ML-based:** ML-based approaches [4, 28, 30, 33, 35] train machine learning models with historical data offline for further predicting proper cache sizes according to the workloads online. However, the pre-trained models based on historical data can hardly be adapted to dynamic online workload scenarios which have quite different data access patterns.
- **MRC-based:** MRC-based methods [15, 19, 36, 52] explore the optimal cache size by generating a miss ratio curve (MRC) as a function of the cache size. To reduce the overhead of generating MRCs, several approaches [22, 40, 41, 45] use sampling techniques. However, they are likely to miss heavy hitters, which would incur inaccuracy. Moreover, most MRC-based approaches are designed for fixed-size item access, which might be inaccurate for variable-size item.
- **Window-based:** Window-based methods [5, 11, 20] estimate the cardinality of items in a sliding window. However, they can hardly compute the accurate total size due to being unaware of each item's size with limited memory space and little time cost.

3 Design of Cuki

To efficiently estimate the real-time working set size (WSS) and the item repetition ratio (IRR) of various-granularity data access over sliding time windows, we design a compact approximate data structure called Cuki. In addition, we have theoretically proved that Cuki outperforms the state-of-the-art comparing algorithms in space usage under the same false positive rate. The proof details are moved in Appendix B due to page limitation. In this section, we introduce the main data structure and supported operations of Cuki.

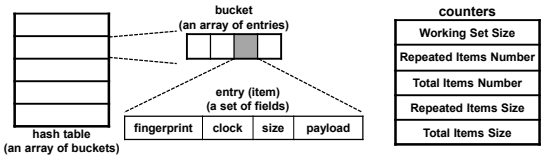


Figure 1: Data structure of Cuki.

3.1 Data Structure

Cuki is built on the Cuckoo filter [17]. The first reason we choose the Cuckoo filter is that it supports deletion so that we can remove stale items. Second, different from the Bloom filter, one item occupies one slot in the Cuckoo filter, so it is easy to extend cells for recording items' size.

In general, Cuki is an approximate membership query data structure with the following features: 1) similar to the Cuckoo filter [17], Cuki stores the items' fingerprints rather than the original data, which is memory-efficient. Different from the Cuckoo filter, Cuki has a more sophisticated design to support time window semantics, working set size estimation, and payload field extension. 2) Cuki supports insertion, lookup, deletion, and aging operations at the item level with efficient concurrency access control mechanisms. 3) Cuki provides built-in efficient and accurate working set size estimation in multi-scopes over the sliding time window based on the lightweight tracking of each item insertion and deletion.

Figure 1 shows the data structure of Cuki. It contains a *hash table* with multiple fixed-length *buckets*, each of which has several fixed-length *entries* to store items. Each entry has four fields to track an item's information: fingerprint, clock value, encoded size, and payload. In addition, there are five kinds of atomic *counters*, including *working set size*, *repeated items number*, *total items number*, *repeated items size*, and *total items size*, which are high-level global metrics. Particularly, *repeated items number* records the number of items that are repeatedly accessed in a sliding window, and the *total items number* is the total accessed items number in a sliding window. Cuki can calculate IRR by simply dividing *repeated items number* by *total items number*. All these five metrics are updated along with inserting or deleting items. Similar to the IRR, the bytes repetition ratio can also be easily calculated as *repeated items size* / *total items size*.

The **fingerprint** field is a succinct representation of an item. Usually, the fingerprint has few bits and is much smaller than the original item size. Moreover, similar to the Cuckoo filter, the fingerprint length of Cuki also offers a trade-off between accuracy and space, *i.e.,* Cuki can

achieve more accurate estimation with longer fingerprints.

The **clock** value represents the freshness of an item. The higher, the fresher. Cuki sets the clock value of an item to a predefined value of MAX_AGE when the item is accessed (insertion or lookup) and periodically decreases it over time by the aging operation. Using s bits for each clock, MAX_AGE is set to $2^s - 1$, where s is an accuracy-to-space trade-off parameter. Suppose the sliding window size is $\mathcal{T}$, the aging operation will be executed every $\frac{\mathcal{T}}{2^s-1}$. The aging operation ensures that the stale items are cleared timely. Moreover, since the aging operation is executed more frequently using a longer clock bits length, there will be fewer errors in sliding windows. Aging operations can work in the background. In addition, we can further amortize the computation overhead by the opportunistic aging strategy in § 4.2.

The **size** field stores the encoded size of each item. There exist some naive several size encoding techniques, such as **Full-size Encoding** which directly stores each item's exact accurate size and **Truncation Encoding** that only stores the higher bits of the item size, since they are more important than the lower bits. To make a better tradeoff between accuracy and space overhead, we propose the **Grouped Size Encoding** technique. It saves the lower bits of each item into size groups. Every prefix has a corresponding size group to record the size of items with the same prefix. Each size group has two counters: *counts* (total number of items) and *total_bytes* (total item size). For insertion, Cuki increases *counts* by 1 and *total_bytes* by the item's size. When an item is removed, Cuki decreases *total_bytes* by the average size *total_bytes*/*counts*, and *counts* by 1. For a prefix length of $\gamma \cdot len$ bits, there are $2^{\gamma \cdot len}$ size groups in total. The space overhead of grouped size encoding is $\gamma \cdot N \cdot len + 2^{\gamma \cdot len} \cdot C$, where C is the bits length of the above two counters for each size group.

Apart from the above size encoding methods, more sophisticated size encoding strategies [6, 7, 14] are also compatible with Cuki. However, these methods require additional computation. Thus, we choose not to use them as the main strategies.

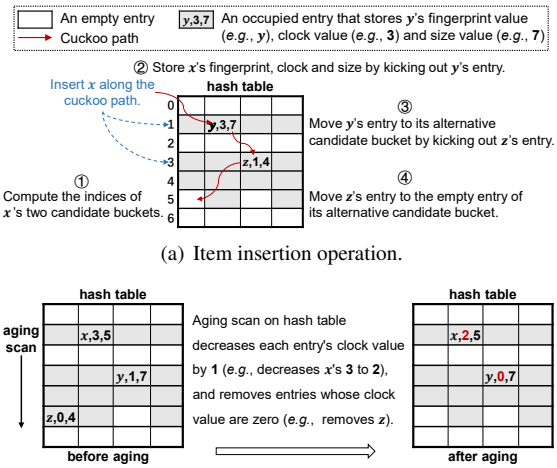
The **payload** field stores auxiliary information of an item. Although the former three fields are enough for the working set size estimation problem, we leave the payload as an auxiliary field for customized needs. In § 3.2, we introduce an example extension usage of the payload field, namely the multi-scope working set size estimation.

3.2 Operations in Cuki

Item Insertion: First, Cuki computes the fingerprint and two bucket indices b_1 and b_2 of a given item x by Equations (1) ~ (3), respectively.

$$f = fingerprint(x), \quad (1)$$
$$b_1 = hash(x), \quad (2)$$
$$b_2 = b_1 \oplus hash(f). \quad (3)$$

Through Equations (1) ~ (3), Cuki can compute two candidate buckets by fingerprint without original item information. Next, it searches for an empty entry in the two candidate buckets. If successful, Cuki stores the item's fingerprint and encoded size in that entry and initializes the entry's clock value to MAX_AGE. Otherwise, it relocates other entries iteratively until it finds an empty entry. Specifically, Cuki finds a cuckoo path in the hash table [18]. The cuckoo path starts with a candidate bucket and ends with an empty entry. Cuki performs the insertion by moving items along this cuckoo path. For example, the red color line in Figure 2(a) is a cuckoo path, which starts with the bucket 1 and ends with the bucket 5. For inserting x into bucket 1, the entry y in bucket 1 will be kicked out to bucket 3. This leads the entry z in bucket 3 will be kicked out to bucket 5. The Cuckoo path length will grow as item insertion, which might lead to long tail latency. We will discuss how to mitigate this in § 4.1.



(a) Item insertion operation.

(b) Aging operation.

Figure 2: Illustration of insertion/aging operations in Cuki.

Item Lookup: Cuki's item lookup first computes the fingerprint and two bucket indices of a given item x by Equations (1) ~ (3). Then, it checks if there exists an entry that matches the fingerprint within the two candidate buckets. If yes, it resets this entry's clock value to MAX_AGE and returns true. Otherwise, it returns false.

Item Deletion and Aging: Cuki supports removing an entry (item) by the item deletion operation or the item aging operation (§ 4.2) at an item's maximum age. Cuki's deletion method first looks up the candidate buckets which are described above. Then, it removes the entry which matches the fingerprint.

Item updating: If an item's attribute (*e.g.,* size) changes, Cuki needs only one-single table access to swiftly alter the hash table entry's fields without searching the cuckoo path. If an item's ID changes, Cuki considers it a new insertion. The old item can be deleted ad-hoc or via aging with performance-accuracy tradeoff. In addition, data updates are non-common in big data OLAP applications.

It is not trivial to automatically remove stale items from the sliding window. A straightforward accurate solution is to

record all item IDs and timestamps (64 bits). This method requires too much memory because of the large number of timestamps. In recent years, some methods [3,5,20] try removing stale items without timestamps. However, as we analyze in § 6.3, these methods are either poorly memory utilized or inaccurate. Different from these methods, the clock algorithm [13] can remove stale items in time with little memory overhead (8~16 bits, as shown in § 6.2).

Therefore, we introduce *clock* into our data structure. Every entry in Cuki is associated with a clock value. Once an entry's clock value reaches zero, it should be removed because of the staleness. This can be done by periodical aging operations in the background. Specifically, suppose the length of the sliding window is $\mathcal{T}$, the bits length of the clock field is s, then the aging period of Cuki is $\frac{\mathcal{T}}{2^s-1}$. The length of window $\mathcal{T}$ can be either time-based or count-based [11, 20]. The time-based sliding window contains items that arrive in the last $\mathcal{T}$ time units. The count-based sliding window contains the last $\mathcal{T}$ items. In each aging operation, it iterates the whole hash table in order and decreases each entry's clock value by 1. If an entry's clock value is already down to zero before aging, Cuki deletes this entry. Figure 2(b) shows an aging example. For items x and y, the aging operation decreases their clock values by 1; while for item z, whose clock value was zero, it is removed. Though using clock to remove stale items can save much memory, it brings errors in results. We have put the theoretical analysis of the above statement in Appendix B.1 due to space limitation.

Entire Working Set Size Enquiry: Besides the above item-level operations, Cuki also natively supports working size-level operations, such as the entire working set size enquiry.

In fact, computing the entire WSS by online scanning the entries in a full hash table and summing up their sizes is very time-inefficient and resource-costly for each query request. Instead, we maintain a counter inside Cuki. The counter tracks the WSS, updates it when inserting or deleting (*e.g.*, by aging) items, and can thus always answer entire WSS enquiry in constant time. The counter is implemented with an atomic class. Thus, it can be concurrently updated safely and efficiently.

Multi-scope Working Set Size Enquiry: In addition to entire WSS enquiry, Cuki also supports WSS enquiry at the scope level, which queries the sizes of specific scopes of the entire working set. Different scopes can be regarded as different parts of the entire working set (*e.g.*, different tables of a database, or different partitions of a table). The information of each scope size is useful for resource scheduling methods [39] and optimizing multi-tenant systems [23, 46]. For example, in a large-scale query engine, we can use multi-scope WSS estimation to find the table with the biggest WSS, which is usually queried frequently. Replicating this table to more nodes of the cache system may help increase the throughput of the query engine.

To estimate multi-scope WSS, we can easily encode the scope information (*e.g.,* mapping scopes to an integer by a hash table) into several bits and store them in the payload field of Cuki. In addition, we need to maintain a set of independent counters (*e.g.,* WSS, repeated items size, and total items size) for each scope in Cuki. For example, when an item x belonging to scope $Scope_k$ is inserted, Cuki will store the encoded scope $Scope_k$ along with the entry of x, and increase the independent WSS counter of $Scope_k$. When the deletion or aging operation removes x, Cuki can figure out the scope that x belongs to, by checking the encoded scope information in its payload field, and decreases the relevant WSS counter of that accordingly.

In practice, for existing methods, it is non-trivial to allocate a suitable memory size for each scope without prior knowledge of each scope's cardinality. Instead, in Cuki, the items of different scopes can share the same total hash table space, by using the encoded scope information in their payload fields to distinguish from each other. Thus, it is not necessary to allocate static memory space for each scope in Cuki.

4 Concurrency Control in Cuki

4.1 Segmented Lock and Concurrent Insertion

We first introduce the basic concurrency control technique called **segmented lock** adopted in Cuki. Then, we describe how Cuki supports concurrent insertion.

Segmented Lock: Cuki divides the whole hash table into several segments, and each segment is guarded by one single lock. Users can configure the number of buckets per segmented lock to tradeoff lock overhead and contention. On the one hand, the item insertion, lookup, and deletion may access different segments at the same time. To avoid deadlock in operations, we always acquire and release the locks in order. On the other hand, each segmented lock guards a group of adjacent buckets. Therefore, for the aging operation, there is no need to repeatedly acquire a lock for scanning items in the same segment. Moreover, each lock manages a physically continuous space. Benefiting from this cache-friendly design, the aging operation can be executed faster. This is because the aging operation accesses Cuki sequentially.

Concurrent Insertion: It is non-trivial to handle the concurrent insertion operations in Cuki. As analyzed in [18], there will be a false negative error under concurrency when moving items along the cuckoo path. To eliminate the false negative error, similar to [18, 26], we separate the insertion process into two phases: the path discovery and item movement phases. In the path discovery phase, Cuki finds a cuckoo path [18] that starts from two alternative buckets and ends at an empty entry. Then, in the item movement phase, Cuki moves items backward along the cuckoo path. Cuki always acquires locks before each above phase, guaranteeing each operation's atomicity.

With more items inserted into Cuki, the Cuckoo path length increases, which might lead to long tail latency. The item

movement may also fail as analyzed in [26]. The probability of insertion failure is less than 1.75×10^{-5} in their environment. In our experiment and production environment, there is almost no insertion failure most of the time. Also, we find that 97% of Cuckoo paths have lengths below 2, and 99.99% of Cuckoo paths have lengths below 4 in MSR trace with 192KB memory. To totally avoid insertion failure and long tail latency, one can allocate appropriate memory size for Cuki by using space resizing techniques [10, 27, 43, 50]. Specifically, when the load of Cuki reaches the high watermark, according to the solution proposed in [50], we can resize the Cuki's capacity by adding an extra homogeneous Cuki data structure after the existing one. The new incoming items can be inserted into the expanded data structure [50]. Except for this solution, we can adopt "partial-key linear hashing" technique proposed in [43] to increase the capacity of Cuki in a fine-grained fashion. Furthermore, similar to [26], we adopt **breadth-first search** to find an empty entry. It can be theoretically proven that the Cuckoo path found by BFS is shorter than that found by DFS [26].

4.2 Opportunistic Aging

To update the data freshness over the sliding window, Cuki performs aging operations periodically in the background. At each background aging, Cuki scans the whole hash table. It first acquires the lock of each segment, then ages the items in that segment in turn. However, the background aging suffers from the following issues in high concurrency scenarios.

Issue 1. Large fluctuation of estimation result: In background aging, massive obsolete items will be cleared simultaneously. As a result, the estimated working set size varies a lot before and after the aging execution. Therefore, the aging can significantly affect the error in the estimated WSS, which decreases the estimation accuracy and stability. We conduct an experiment to verify this, and it is in Appendix C.1.

Issue 2. Lock contention with user operations: Most operations in Cuki (*e.g.*, insertion, lookup, and aging) require holding the lock first, which causes lock contention among these operations. It brings in two kinds of issues. First, when the aging operation is in execution, if there are too many obsolete items that need to be removed, the other data access operations will be blocked for a long time until the lock held by the aging process is released. It increases the delay of other data access operations. Second, when the aging operation is waiting for execution, if there exist so many insertions or lookup operations, the aging operation might wait a long period before getting the lock. Thus, the obsolete items in Cuki may not be removed in time by aging, which decreases the estimation accuracy of Cuki.

To address these issues, we propose a lightweight concurrency control strategy called **opportunistic aging**. It amortizes the aging operation into the insertion operations in Cuki. It brings two main advantages. First, the full aging task is split into multiple minor aging tasks, making the sliding window move smoother. Second, since fewer entries need to be checked in background aging, it reduces the lock contention risk with the background aging.

Algorithm 1 Opportunistic Aging in Cuki

Input: S_i is the segment to be aged, P_i is the aging pointer of S_i.

1: $N_{oa} \leftarrow$ the number of items to be aged;
2: **while** $N_{oa} > 0$ && $P_i < S_i.length$ **do**
3: /* aging the P_ith buckets of segment S_i */
4: AgingBucket(GetBucket(S_i, P_i));
5: $P_i \leftarrow P_i + 1$;
6: $N_{oa} \leftarrow N_{oa} - 1$;

Specifically, each segment in Cuki has a pointer to track its aging progress. Both opportunistic aging and background aging start working from the pointer's position. N_{oa} items are aged during each opportunistic aging. The pointer advances accordingly. Subsequently, background aging ages the remaining items in each segment from the position of the pointer left by opportunistic aging. If the aging pointer is at the end of the segment, background aging will skip this segment. Therefore, opportunistic aging reduces lock contention.

Algorithm 1 elaborates the procedure of opportunistic aging. First, it computes the number of items that need to be aged (noted as N_{oa}) by the elapsed time from the beginning of the aging period (Line 1). Suppose S_i is the segment to be aged, P_i is the aging pointer (index) of S_i, N_i is the number of buckets in S_i, T is the time interval of each aging period, t_{cur} is the elapsed time from the beginning of the aging period, N_{oa} can be computed by the equation:

$$N_{oa} = N_i \times \frac{t_{cur}}{T} - P_i. \tag{4}$$

It guarantees that the aging progress is consistent with the movement of the sliding window. Besides, to reduce the latency of each insertion operation, we limit the maximum number of items cleared during each opportunistic aging. Then, We conduct the aging operation on the N_{oa} items in the segment S_i (Lines 2-6). The remaining stale items, which have not been removed by opportunistic aging, will be cleared by the background aging.

Regarding accuracy, ClockSketch [11] reveals that some stale items are not cleaned timely by background aging (also analyzed in our Appendix B). Opportunistic aging can mitigate this error by preemptively removing certain stale items before background aging.

5 Cache Capacity Online Tuning Using Cuki

In this section, we show how Cuki can facilitate online cache capacity tuning for many data access applications. First, Cuki can be used in implementing the cache size adaptive tuning mechanism. Based on that, it can accelerate data access, including table querying and file reading. In addition, Cuki can

also help generate miss ratio curves (MRCs), which provides an in-depth understanding of the relationship between the cache hit ratio and the cache size.

5.1 Data Access Application Acceleration

During setting the cache capacity for applications, we are mainly faced with two key cache-related questions: 1) What is the degree of the data access temporal locality for a given data access stream? 2) How to optimize the cache utilization online for a given data access stream?

5.1.1 Adaptive Cache Capacity Tuning Framework

In the following, we introduce the key metrics of the Cuki, which can be used for adaptive cache capacity tuning. To explain how to track and optimize the cache utilization with Cuki, we define the following key metrics.

- **CSS:** The cache space size, which can usually be obtained from configurations or metric monitoring of the cache system.
- **WSS:** The working set size over the time window, which is estimated by Cuki online.
- **CHR:** The realistic cache hit ratio of the cache system, which is often exposed by the cache metric monitor system.
- **IRR:** The item repetition ratio over the time window estimated by Cuki. IRR is computed by $\frac{\|R\|}{\|O\|}$, where O and R are the set of total accessed items and repeatedly accessed items in the time window, respectively.

The proposed adaptive cache capacity tuning mechanism can answer the above questions by tracking WSS and IRR in constant time with Cuki.

IRR measures the data access temporal locality of the application online. Specifically, since every repeatedly accessed item is counted by Cuki, IRR can be regarded as the upper bound of the cache hit ratio for the realistic cache system over the past time window. WSS is the total size of recently accessed items. It reflects the realistic cache demand of the application in the time window. In fact, as we observed in our real-world query service scenarios and other applications reported in existing work [48, 53], the working set size and data repetition ratio do not significantly change in a short period, following the law of temporal data locality. Thus, for a workload, its estimated WSS and IRR over adjacent time windows are likely similar, and we can use the current estimation to optimize the cache capacity for the near future.

Cuki has two main advantages in estimating WSS and IRR. First, it can track the WSS over sliding time windows accurately and stably. Second, it supports updating and querying WSS with constant time complexity, which makes real-time tracking and dynamic adjustment possible.

Figure 3 illustrates how Cuki and the above defined metrics can help to improve the cache system efficiency. Cuki is embedded into cache layer and cooperates with

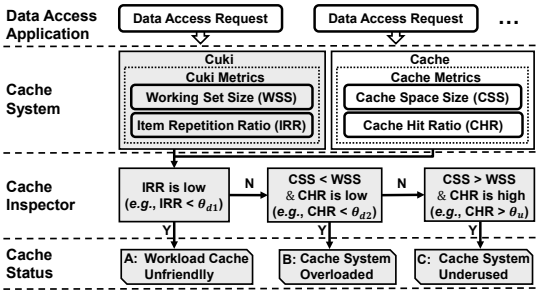


Figure 3: Workflow of adaptive cache capacity tuning mechanism based on Cuki (grey components are proposed by us).

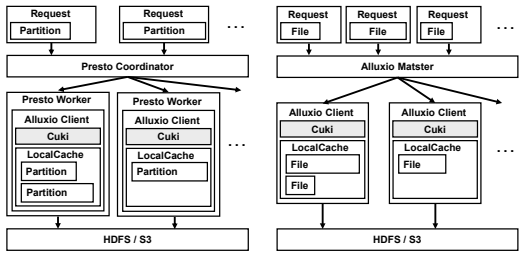
the cache system seamlessly. The cache system has online CSS and CHR metrics, while Cuki contains WSS and IRR statistics during the corresponding time windows. When a data access request arrives, the metrics of the cache system and Cuki are simultaneously updated.

The cache inspector figures out the cache system status by comparing these metrics according to the logic in Figure 3. Based on the cache status and the metrics in Cuki, the cache space size can be appropriately tuned up and down online.

Specifically, the cache inspector will measure the data temporal locality of the workload by checking the item repetition ratio IRR estimated by Cuki. For case A in Figure 3, if IRR is low (smaller than a predefined threshold θ_{d1}, *e.g.*, 50%), the workload itself is not cache-friendly, which means that there exists little repeated data access during the time window. In this case, even if adding huge cache space, we can barely get a higher cache hit ratio CHR.

In other cases, when IRR is high, the cache inspector will check CHR, cache space size CSS, and the working set size WSS over the time window. For case B in Figure 3, when CHR is low (smaller than a predefined threshold θ_{d2}, *e.g.*, 50%) and CSS<WSS, it means that the cache system is overloaded, and there indeed exists some room to improve the cache performance further. This is because the CHR is low, but CSS is still less than the realistic cache demand measured by the estimated WSS. In this case, we can improve the cache efficiency by increasing CSS. For example, the size of an application's data table usually increases as the number of application users grows. The cache system will be under-provisioned if CSS is not carefully configured accordingly. However, with the estimated WSS as the indicator, we can allocate an appropriate amount of cache resource easily online.

Besides, if both IRR and CHR are high (larger than a predefined threshold θ_u, *e.g.*, 90%), it means that the cache system has sufficient cache resource. However, the resource might be wasted when we allocate superfluous cache space over the realistic cache demand measured by the estimated WSS in Cuki (case C in Figure 3). In this case, the cache system is underused. In real-world practice, we can tune down the cache space or leverage this information to optimize the query task scheduling algorithms or the cluster resource routing strategies. For example, we can facilitate the load balance of the cache system by prioritizing scheduling the query tasks



(a) Table querying framework (b) File reading framework

Figure 4: Data access applications using Cuki.

to the compute nodes where the cache is still underused.

In the last case, where both the cache hit ratio CHR and the item repetition ratio IRR are high, and the cache space size CSS matches the estimated realistic cache demand WSS, the cache system is working in healthy status.

5.1.2 Table Querying Acceleration Framework

Figure 4(a) shows how the proposed adaptive cache capacity tuning mechanism is integrated with Presto and Alluxio. Presto is designed for performing SQL query computation in memory. The Presto coordinator distributes the execution plan fragments to Presto workers according to the scheduling strategy. Presto workers execute query plan fragments on the data read from the remote HDFS/S3. Since Presto workers do not store the data, they tend to use Alluxio clients as their cache tier. We implement the Cuki in Alluxio client to track the LocalCache access. In order to use Alluxio LocalCache as Presto worker's cache, Alluxio client Jar files are distributed to each Presto worker.

Each Presto worker queries Alluxio LocalCache inside the same JVM through a standard HDFS interface. First, Presto transforms the queried partitions into several splits. Then, Presto coordinator makes the best attempt to assign the same split to the same worker, which is cache-friendly. If the queried splits are in the Alluxio LocalCache, splits are directly read from local RAM and returned to Presto. Otherwise, it retrieves data from HDFS/S3 and caches the data to local RAM of Presto worker. Cuki monitors the whole process of the split access in each Presto worker and updates WSS/IRR.

5.1.3 File Reading Acceleration Framework

File reading is common in distributed applications, such as online video websites and cloud downloading services. It's common that there exist some hot files which are more likely to be accessed by applications in a nearby time period. Thus, we can use a cache system to accelerate file reading by storing hot files. However, the size of hot files changes as time flies, which makes it hard to determine the proper cache size. As shown in Figure 4(b), the proposed adaptive cache capacity tuning mechanism based on Cuki can be used to solve this problem. The implementation of the proposed adaptive cache capacity tuning mechanism in file reading is similar to the above section. First, the requests for files are sent to the Alluxio master. Then, the Alluxio master checks whether the requested files are stored in one of the Alluxio clients. If so, the requested files are directly read from the local RAM of the Alluxio client. Otherwise, Alluxio reads files from HDFS/S3 and caches the data to the local RAM of the Alluxio client. Cuki monitors the whole file access process in each Alluxio client and updates WSS and IRR accordingly online.

5.2 Miss Ratio Curves Generation Using Cuki

Although WSS and IRR are useful enough for the adaptive cache capacity tuning mechanism, they still can not show in-depth insights into the relationship between the cache hit ratio and the cache size. Generating a miss ratio curve (MRC) as a function of the cache size is a common method to understand the relationship between the cache hit ratio and the cache size thoroughly. It only needs a little change in Cuki to generate MRCs for variable-size item access.

Similar to most MRC generation approaches [40, 41, 51], Cuki needs to store the reuse distance distribution as $RD(x)$. $RD(x)$ represents how many items are re-accessed at x LRU stack size (x is also called as the reuse distance). With $RD(x)$, Cuki can compute $MRC(x)$ simply by $\frac{\sum_{i=1}^{x} RD(i)}{TC}$, where TC is the total items number. To compute $RD(x)$, Cuki tracks each item's clock value and stores the clock distribution as $CD(y)$. $CD(y)$ represents the total size of items whose clock value is y. With CD, Cuki computes the reuse distance of the accessed item by $distance = \sum_{i=y}^{max} CD(i)$, where y is the clock value of the accessed item. Then, Cuki increases the $RD[distance]$ by one. The length of the array CD is the MAX_AGE, which is decided by the clock bits length. Because the clock bits length is a small constant number (never exceeding 16 in our evaluation), the space cost of CD is negligible.

In the following, we introduce how Cuki maintains CD when the item's clock value changes. We use oc and nc to represent the old clock value and the new clock value, respectively. The item's clock value changes when the item is accessed or aged. Then, Cuki decreases the $CD[oc]$ by the item size and increases the $CD[nc]$ by the item size.

6 Evaluation

6.1 Experimental Setup

To be consistent with Alluxio and Presto, we implement Cuki and comparison methods in Java. If not explicitly mentioned, all approaches run on a server with Intel Xeon(R) Gold 6248 CPU with ten 2.5GHz cores. The version of Alluxio and Presto in the experiment is 2.7.0 and 0.266, respectively.

Datasets and Workloads. Experiments are run on both existing benchmarks and real-world datasets with workloads:

(1) **MSR I/O trace dataset [31].** We choose the first 12,518,968 records of MSR web proxy workload

as a typical dataset. Each record consists of three disk access information: timestamp, offset, and size. In our experiment, we use the offset to represent the item ID.

(2) **Twitter dataset [48]**. We choose the representative Twemcache-cluster37 first-hour data which has 10,169,267 records, and the similar Twemcache-cluster35 first-day data as our datasets. The record's key is regarded as the item ID, and the item's size is the sum of key_size and value_size.

(3) **YCSB dataset [12]**. We generate a concatenated trace that contains 10 million records by the YCSB generator [47]. Each base trace follows a zipfian distribution [32] with a skewness factor of 0.99. The item size of each base trace ranges from 512B to 1MB, but follows different zipfian distributions.

(4) **TPC-DS [38]**. Typical I/O bound queries in TPC-DS are used for the end-to-end performance evaluation.

(5) **Real-world query workloads**. We also adopt the real-world query workloads from one of our large scale Presto clusters with 200 servers in § 6.8. The total data access size of the workload is PB-level, and the cache space is TB-level.

Comparison Approaches. Following methods are evaluated:

(1) **ClockSketch [11]**. We add a 32-bit size counter for each cell of ClockSketch for WSS estimation. When a cell is first inserted, its size counter will be set to the item's size.

(2) **SlidingSketch [20]**. We apply SlidingSketch to the Bloom filter [8] for WSS estimation. Each domain of its bucket is used to record the item size.

(3) **SWAMP [5]**. SWAMP stores each item's frequency in a data structure called TinyTable [16]. We extend the TinyTable in SWAMP to record each item's size.

(4) **MBF [3]**. We use the Multiple Bloom Filter (MBF) implementation in the latest Alluxio version [3]. Each Bloom filter [8] is implemented with Google's Guava library [2].

(5) **RAR-CM [51]**. In RAR-CM, each block has a counter to record the last access number. To support variable-size item, we use the RAR-CM's counter to record its last access bytes.

(6) **Cuki and Cuki-OA**. Cuki is the basic approximate data structure proposed in this paper. Cuki-OA further uses the opportunistic aging strategy in § 4.2.

Metrics. We measure accuracy and speed performance by following metrics:

(1) **Weighted Error Rate (WER)**. Let *error_bytes* be the total size of items that are evicted faster or slower than the ideal sliding window. The WER can be calculated by $\frac{error\_bytes}{total\_bytes}$.

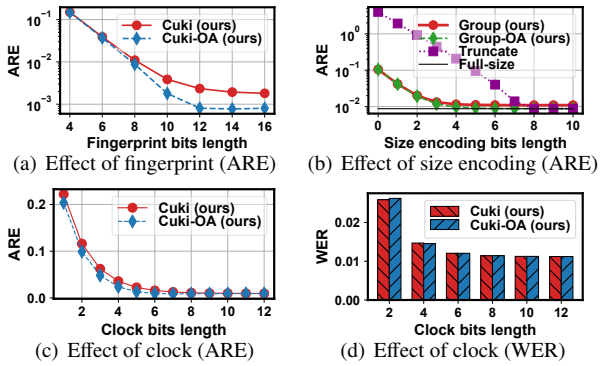
(2) **Relative Error (RE)**. $\frac{w-\hat{w}}{w}$, where w and $\hat{w}$ are exact and estimated working set size (WSS), respectively.

(3) **Average Relative Error (ARE)**. $\frac{1}{|T|}\Sigma_{t\in T}\frac{|w_t-\hat{w}_t|}{w_t}$, where w_t and $\hat{w}_t$ are the exact and estimated WSS at moment t.

(4) **Mean Absolute Error (MAE)**. $\frac{1}{N}\sum|MRC(x)-MRC'(x)|$, where N is the length of reuse distance array, $MRC(x)$ is the hit ratio at the cache size x.

(5) **Throughput**. In file reading experiment (§ 6.7), it is the number of MB/s. In other experiments, it is # of operations/s.

(6) **Query Latency**. The end-to-end SQL query latency.



(a) Effect of fingerprint (ARE) (b) Effect of size encoding (ARE)
(c) Effect of clock (ARE) (d) Effect of clock (WER)

Figure 5: Effects of parameters in Cuki.

Parameter Settings of Approaches. All methods use the same memory size in each experiment. For the count-based sliding window, we set the window size to 262,144 (2^{18}) and measure RE every 64 time units. For the time-based sliding window, we set the window size to one-hour and one-day for the MSR and Twitter traces as different traffics, respectively. The default size encoding approach for Cuki is grouped size encoding. The bits length of the fingerprint, clock and size fields in Cuki are set to 8 if not explicitly mentioned. The settings of the comparing methods are fully tuned to nearly achieve their best performance for a fair comparison.

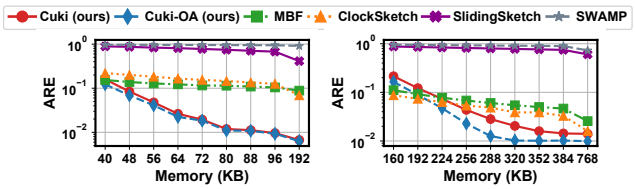
6.2 Effect of WSS Estimation Parameters

To understand the impact of the Cuki's parameters, we conduct experiments on the YCSB trace with a count-based sliding window. The number of entries in Cuki is fixed to 262,144 (2^{18}), just enough to track all the items within a sliding window. To reduce other parameter interference, we use the full-size encoding method by default in this section.

(1) **Effect of fingerprint bits length.** As shown in Figure 5(a), as the fingerprint bits length grows, the ARE of both Cuki and Cuki-OA is dramatically decreased. In fact, an item's key is represented by its fingerprint. Thus, a small fingerprint bits length leads to different items being hashed to the same fingerprint, resulting in high ARE. Moreover, compared with Cuki, Cuki-OA decreases ARE by 37% on average, which verifies the effectiveness of the opportunistic aging strategy.

(2) **Effect of size encoding methods.** Figure 5(b) illustrates the influence of different size encoding methods. The performance of the truncation encoding method using and not using opportunistic aging is the same. Thus we only show the truncation encoding in the figure. The black line represents the ARE of the most accurate baseline (full-size encoding).

As shown in Figure 5(b), on the one hand, full-size encoding achieves the best accuracy but it stores the entire accurate size. Compared with the truncation encoding strategy, the grouped size encoding strategy decreases ARE by 92% on average when the group bits length is small (< 6 bits). Thus, we can conclude that grouped size encoding achieves the best trade-off between memory space and estimation accuracy.



(a) ARE on MSR trace (b) ARE on Twitter trace

Figure 6: Performance comparison of accuracy.

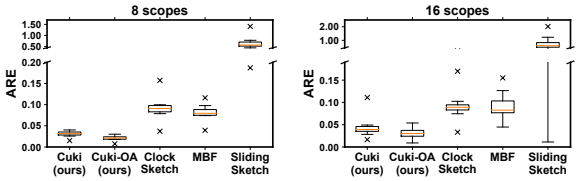


Figure 7: Accuracy of multi-scope estimation.

(3) **Effect of clock bits length.** As shown in Figures 5(c) and 5(d), the ARE and WER can be reduced by using more clock bits. As shown in Figure 5(c), opportunistic aging (Cuki-OA) decreases the ARE of Cuki by 26% on average when the length of the clock bits is small (< 8 bits). Also, we can observe from Figure 5(d) that Cuki-OA barely increases WER.

Besides the above three parameters, the parameter sliding window size can be set as the user demands. In the above experiments, Cuki only needs a few extra bits to track each item's key, access freshness, and size. Therefore, we can conclude that Cuki can track each item only using several bits by sacrificing negligible accuracy.

6.3 Accuracy of WSS Estimation

In this subsection, we evaluate the accuracy of Cuki by comparing it with cutting-edge WSS estimation methods over the sliding window mechanism. Figure 6 exhibits the ARE of different methods measured in the same run on two traces. We double the memory size at the last point of each experiment to meet the memory requirement of each approach. As shown in Figure 6, while the performance of all comparison approaches gets improved with more space, Cuki and Cuki-OA exhibit better memory-accuracy efficiency. For example, Cuki-OA decreases ARE from 12.26% to 0.93% as the memory space increases to 96KB on the MSR trace. In addition, Cuki-OA decreases the ARE of Cuki by an average of 11% and 37% on the MSR and Twitter traces, respectively. As the memory space gradually becomes larger, the ARE of Cuki decreases to 1% and lower. However, the ARE of coarse-grained tracking methods, such as MBF, SlidingSketch, and ClockSketch, can hardly further decrease even with sufficient memory.

Finally, we compare the accuracy of various methods on multi-scope WSS estimation. We use the MSR dataset as a typical benchmark and replay it with a 144× speedup. The time-based sliding window size is set to one hour. All methods in the experiments use the same 24MB memory size because of the large multi-scope combined workload. As shown in Figure 7, Cuki-OA reduces the ARE of Cuki by 33% and 22%

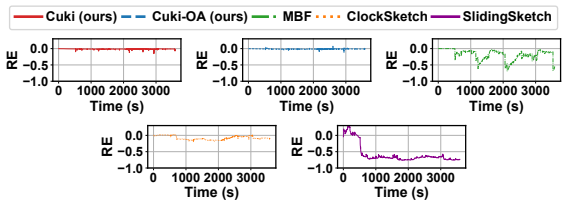


Figure 8: Performance comparison of stability on MSR trace (comparison methods use 2× larger memory than Cuki).

on average for the 8 and 16 scopes WSS estimation, respectively. Secondly, the ARE of Cuki-OA is 11 and 8 times lower than the comparison algorithms on average for the 8 and 16 scopes WSS estimation, respectively. It mainly benefits from Cuki's extensibility which allows items of different scopes to make better usage of memory together.

To conclude, Cuki and Cuki-OA achieve the best accuracy with the same memory consumption among all the methods. More experiments on the YCSB trace or using the WER metric are in Appendix C.2. They have similar conclusions.

6.4 Stability Performance of WSS Estimation

We evaluate the stability performance of different methods under the time-based sliding window. More experiments on the Twitter trace and the count-based sliding window are available in Appendix C.3. They have similar conclusions.

We replay the MSR trace with 168× speedup and use 192KB memory for the Cuki. In order to meet the comparison methods' memory requirements, they use double amount of memory than Cuki. Figure 8 illustrates the stability performance of different methods over the time-based sliding window. SWAMP is omitted due to it only supports the count-based sliding window. There are jagged fluctuations in estimation for all methods because of the movement of the sliding window. Specifically, MBF switches a Bloom filter out periodically and drops the corresponding items. ClockSkech's fluctuations are mainly due to hash collision with limited memory. SlidingSketch can hardly track all items within a sliding window due to limited memory space.

For Cuki, despite its performance being affected by aging operations, its estimation is stable. The stability is mainly attributed to its per-item size tracking. Notably, opportunistic aging can make the movement of sliding windows smoother.

To conclude, Cuki and Cuki-OA use less memory and achieve the most stable estimation results.

6.5 Scalability of WSS Estimation

We evaluate the thread scalability of the comparison methods. Specifically, we use the MSR trace with the count-based sliding window, and the memory size is set to 40KB. SWAMP is omitted due to not supporting multi-thread concurrency.

Figure 9 shows that increasing concurrency can not improve ClockSketch's throughput significantly. SlidingSketch

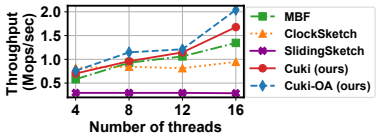
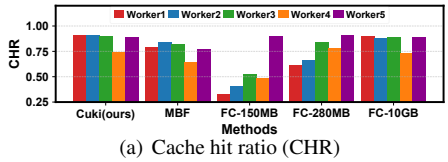
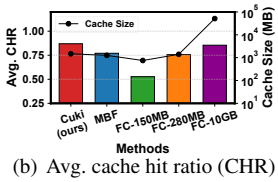


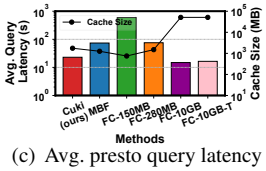
Figure 9: Throughput with concurrent threads.



(a) Cache hit ratio (CHR)



(b) Avg. cache hit ratio (CHR)



(c) Avg. presto query latency

Figure 10: Performance of adaptive cache capacity tuning mechanism in table querying (**FC-100MB** represents the fixed cache size 100MB, **FC-10GB-T** represents running with WSS estimation in the fixed cache size 10GB, others are similar).

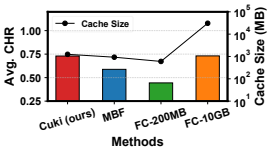
has heavy aging tasks after each operation. Thus, its scalability is limited. MBF can improve throughput by increasing concurrency, but it needs to manage Bloom filters for insertion and lookup, resulting in lower throughput than Cuki.

Cuki and Cuki-OA have near-linear multi-threading scalability due to their fine-grained concurrency control optimization strategies. To conclude, both Cuki and Cuki-OA achieve near-linear multi-threading scalability.

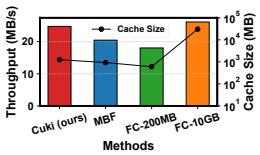
6.6 Cache Tuning Performance with Cuki

(1) **End-to-End Performance in Table Querying :** The experiments run on a Presto cluster with one coordinator and five workers using the I/O-bound TPC-DS dataset. FC-150MB, FC-280MB, and FC-10GB represent the cache system is in overloaded, healthy, and underused statuses, respectively. The 280MB cache size is manually chosen because it is the most competitive cache size that makes a good trade-off between the cache hit ratio and the cache capacity.

The adaptive cache capacity tuning mechanism use 125MB memory, which is the default value in MBF [3]. As shown in Figure 10(a), the cache hit ratio of FC-150MB is the lowest one. And, the cache hit ratio of FC-10GB can be regarded as the upper bound. By using Cuki, the cache system nearly achieves the upper bound of the cache hit ratio. Figure 10(b) shows the average cache hit ratio and the maximum total cache space allocated by the proposed adaptive cache capacity tuning mechanism and others. Compared with FC-280MB, our method improves the average cache hit ratio by around 11% while using a similar total cache size. This is because that our method allocates the cache space to each Presto worker according to their different demand. Overall, by using Cuki, the cache system can not only reach the upper limit of the

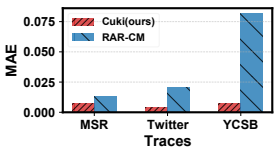


(a) Avg. cache hit ratio (CHR)

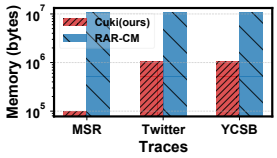


(b) File reading throughput

Figure 11: Performance of adaptive cache capacity tuning mechanism in file reading (**FC-200MB** and **FC-10GB** represent the fixed cache size are 200MB and 10GB).



(a) Mean absolute error



(b) Memory

Figure 12: Performance of Cuki in MRCs generation.

cache hit ratio, but also improve the cache utilization.

The average query latency of different approaches is shown in Figure 10(c), the average query latency of FC-10GB-T (the fixed 10 GB cache size with WSS estimation) is close to FC-10GB. The average query latency of Cuki is close to the FC-10GB which is the lower bound of latency. Compared with MBF, FC-150MB, and FC-280MB, Cuki can reduce the query latency by around 69%, 97%, and 71%, respectively.

(2) **End-to-End Performance in File Reading:** This experiment uses the first 9000 data access requests in YCSB [47] trace as the workload. For each unique trace item, we generate a file whose size is the item value and store the file in remote storage S3. We run the experiments on an Alluxio cluster with three EC2 servers and deploy an EC2 client which runs in three threads to access data. Each thread sends 3000 file reading requests and repeats three times.

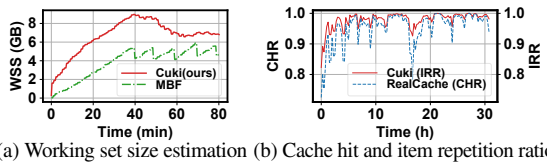
As shown in Figure 11(a) the cache hit ratio of FC-200MB is the lowest. The cache hit ratio of FC-10GB can be seen as the upper bound because the 10GB cache size is enough to cover all workloads. The cache hit ratio of Cuki is close to the FC-10GB, which means Cuki helps the cache system to reach almost the upper bound of the cache hit ratio.

As shown in Figure 11(b), we compare the end-to-end file reading throughput of the above comparison methods. The throughput of Cuki is close to FC-10GB which is the upper bound of the throughput. Overall, Cuki can improve the cache utilization of file reading to reach higher throughput.

6.7 Accuracy of Miss Ratio Curves Generation

We compare the accuracy of miss ratio curves (MRCs) generation among Cuki and RAR-CM. Considering the poor support for the sliding window mechanism in RAR-CM [51], the window length is the same as the trace length.

As shown in Figure 12(b), Cuki uses 96KB memory for MSR trace and 1MB memory for other traces. Each item in RAR-CM needs 128 bits to be stored, which is larger than Cuki's 56 bits. In order to make RAR-CM more accurate, we



(a) Working set size estimation (b) Cache hit and item repetition ratios

Figure 13: Large-scale real-world practice.

allocate RAR-CM 10 MB memory, which is 10× larger than Cuki. Figure 12(a) shows the accuracy of Cuki in MRCs generation. Compared with RAR-CM, Cuki reduces the MAE by around 48%, 82%, and 91% in the MSR, Twitter, and YCSB traces, respectively. This is because that Cuki can better support variable-size item. Moreover, RAR-CM estimates the re-access ratio to compute the reuse distance, which is inaccurate. In addition, Cuki achieves comparable throughput with RAR-CM in experiments. Overall, Cuki costs less memory and generates more accurate MRC than RAR-CM.

6.8 Real-world Practice

We elaborate on how Cuki is used in our real-world large-scale query platforms with the cache system called ShadowCache. ShadowCache is being leveraged to understand the system bottleneck and help with query system routing design decisions. Specifically, with ShadowCache, the overall system can efficiently decide how to size the cache for each tenant, and what the potential cache hit ratio improvement is. In the following, we evaluate the usability of the working set size estimation methods on a middle-scale Presto cluster (GB-level cache space). We implement the proposed Cuki-based cache capacity tuning mechanism. MBF is also used for comparison.

Figure 13(a) shows the estimated WSS of Cuki and MBF on a middle-scale cluster. There exist fluctuation for MBF in its estimation due to periodically removing a part of its statistics as analyzed in § 6.3 and § 6.4. In fact, the cache system can hardly distinguish the normal workload changes from the MBF fluctuations. In contrast, Cuki provides stable working set size estimation with little fluctuation. Thus, Cuki is more credible and effective in real-world scenarios.

Next, we deploy the proposed approach on a large-scale real-world Presto cluster (TB-level cache space with 200 servers). Figure 13(b) shows the performance of query workloads over one day on the Presto cluster, showing the realistic cache hit ratio (CHR) performance of the cache system and the item repetition ratio (IRR) estimated by Cuki. It can be seen that IRR is much higher than the CHR of cache between the 16th hour to the 19th hour. We can find that there is an opportunity to increase the cache capacity based on the estimated WSS during that period to improve the cache hit ratio.

Another interesting discovery during our deployment is that the WSS of each Presto worker is quite unbalanced. This is because that the data hotness of each table or partition is different in real-world scenarios. The extent of the imbalance is related to the access patterns. Cuki is very helpful for global cache space allocation with multiple-scope optimization.

7 Related Work

A key challenge for improving cache utilization is provisioning the suitable cache size to fit dynamic workloads online. As analyzed in § 2, we summarize the prior works in four categories: Rule-based approaches [21, 24, 34, 42], ML-based approaches [4, 28, 30, 33, 35], MRC-based approaches [15, 19, 22, 36, 40, 41, 45, 51, 52], and window-based approaches [5, 11, 20].

The most recent works related to ours are ClockSketch [11], RAR-CM [51], and MBF [3]. ClockSketch [11] maintains a clock value for each item to support the sliding window mechanism. However, ClockSketch uses the bitmap [44] or the Bloom filter [8] to estimate cardinality. It brings WSS estimation error as not being aware of items' various sizes but using maximum likelihood estimation with inferior ARE. RAR-CM [51] uses a hashmap to record item access information and estimate the item repetition ratio. However, RAR-CM is designed for fixed-size item tracking and might be inaccurate for variable-size item tracking. Moreover, RAR-CM has non-negligible memory consumption when handling a large number of unique items. MBF [1, 25] uses a series of Bloom filters to record different statistics in segments of the sliding window. However, the switching of Bloom filters makes the estimation result accuracy unstable.

8 Conclusion and Future Work

In this paper, we propose Cuki, an approximate data structure for estimating the online WSS and IRR for variable-size item access with proven accuracy guarantee. Cuki can also be extended to solve the multi-scope WSS tracking problem. Experimental results show that Cuki outperforms the cutting-edge algorithms by 10× in accuracy. Moreover, the proposed adaptive cache capacity tuning method based on Cuki can significantly improve the cache performance online.

In the future, we plan to explore more application scenarios of Cuki in the cloud-native data processing environment.

Acknowledgements

We thank reviewers and shepherd for their valuable comments and help. This work is funded in part by the National Natural Science Foundation of China (No.62072230, 62272223), Jiangsu Province Science and Technology Key Program (No.BE2021729), the Postgraduate Research & Practice Innovation Program of Jiangsu Province (No.KYCX22_0152), the Fundamental Research Funds for the Central Universities (No.020214380089, 020214380098, 020214912216), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Bin Fan, Haipeng Dai, Rong Gu, and Guihai Chen are the corresponding authors of this paper.

References

[1] Alluxio. https://www.alluxio.io/, 2016.

[2] Guava: Google Core Libraries for Java. https://github.com/google/guava, 2020.

[3] The Implementation of Multiple Bloom Filter. https://github.com/Alluxio/alluxio/blob/v2.7.0/core/client/fs/src/main/java/alluxio/client/file/cache/CacheManagerWithShadowCache.java, 2021.

[4] 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'17)*, pages 469–482, 2017.

[5] Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. Pay for a Sliding Bloom Filter and Get Counting, Distinct Elements, and Entropy for Free. In *37th IEEE International Conference on Computer Communications (INFOCOM'18)*, pages 2204–2212. IEEE, 2018.

[6] Ran Ben Basat, Michael Mitzenmacher, and Shay Vargaftik. How to Send a Real Number Using a Single Bit (And Some Shared Randomness). In *48th International Colloquium on Automata, Languages, and Programming, (ICALP'21)*, pages 439–458, 2021.

[7] Ran Ben-Basat, Gil Einziger, and Roy Friedman. Give me some slack: Efficient network measurements. In *43rd International Symposium on Mathematical Foundations of Computer Science, (MFCS'18)*, pages 543–559, 2018.

[8] Burton H Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proceedings of the VLDB Endowment (VLDB'17)*, 10(12):1718–1729, 2017.

[10] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. The Dynamic Cuckoo Filter. In *25th IEEE International Conference on Network Protocols (ICNP'17)*, pages 1–10, 2017.

[11] Peiqing Chen, Dong Chen, Lingxiao Zheng, Jizhou Li, and Tong Yang. Out of Many We are One: Measuring Item Batch with Clock-Sketch. In *48th ACM Conference on Management of Data (SIGMOD'21)*, pages 261–273, 2021.

[12] 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.

[13] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[14] L De La Peña-Auerbach. A simple derivation of the schroedinger equation from the theory of markoff processes. *Physics Letters A*, 24(11):603–604, 1967.

[15] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *24th ACM conference on Programming language design and implementation (PLDI'03)*, pages 245–257, 2003.

[16] Gil Einziger and Roy Friedman. Counting with Tinytable: Every Bit Counts! In *17th International Conference on Distributed Computing and Networking (ICDCN'16)*, pages 1–10, 2016.

[17] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo Filter: Practically Better than Bloom. In *10th ACM Conference on Emerging Networking Experiments and Technologies (CoNext'14)*, pages 75–88, 2014.

[18] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and Concurrent Memcache with Dumber Caching and Smarter Hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 371–384, 2013.

[19] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event (ASPLOS'21)*, pages 386–400, 2021.

[20] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. Sliding Sketches: a Framework Using Time Zones for Data Stream Processing in Sliding Windows. In *26th ACM Conference on Knowledge Discovery and Data Mining (KDD'20)*, pages 1015–1025, 2020.

[21] Rong Gu, Kai Zhang, Zhihao Xu, Yang Che, Bin Fan, Haojun Hou, Haipeng Dai, Li Yi, Yu Ding, Guihai Chen, and Yihua Huang. Fluid: Dataset Abstraction and Elastic Acceleration for Cloud-native Deep Learning Training Jobs. In *38th IEEE International Conference on Data Engineering (ICDE'22)*, pages 2182–2195, 2022.

[22] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic Modeling of Data Eviction in Cache. In *27th USENIX Annual Technical Conference (USENIX ATC'16)*, pages 351–364, 2016.

[23] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-Driven Multi-Tenant Stream Processing. In *9th ACM Symposium on Cloud Computing (SoCC'18)*, pages 249–262, 2018.

[24] 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'18)*, pages 427–444, 2018.

[25] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *5th ACM Symposium on Cloud Computing (SoCC'14)*, page 1–15, 2014.

[26] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *9th European Conference on Computer Systems (EuroSys'14)*, pages 1–14, 2014.

[27] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard T. B. Ma, Xueshan Luo, and Bangbang Ren. The Consistent Cuckoo Filter. In *38th IEEE International Conference on Computer Communications (INFOCOM'19)*, pages 712–720, 2019.

[28] 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 *31st USENIX Annual Technical Conference (USENIX ATC'20)*, pages 189–203, 2020.

[29] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[30] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: An Opportunistic Caching System for FaaS Platforms. In *16th European Conference on Computer Systems (EuroSys'21)*, page 228–244, 2021.

[31] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage (TOS'08)*, 4(3):1–23, 2008.

[32] David MW Powers. Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*, 1998.

[33] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *12nd International Conference on Cloud Computing (CLOUD'19)*, pages 33–40, 2019.

[34] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A transparent auto-scaling cache for serverless applications. In *12nd ACM Symposium on Cloud Computing (SoCC'21)*, pages 122–137, 2021.

[35] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload Autoscaling at Google. In *15th European Conference on Computer Systems (EuroSys'20)*, pages 1–16, 2020.

[36] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic Performance Profiling of Cloud Caches. In *5th ACM Symposium on Cloud Computing (SoCC'14)*, pages 1–14, 2014.

[37] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. Presto: SQL on Everything. In *35th IEEE International Conference on Data Engineering (ICDE'19)*, pages 1802–1813. IEEE, 2019.

[38] TPC-DS Benchmark. http://www.tpc.org/tpcds/, 2006.

[39] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is Advance Knowledge of Flow Sizes a Plausible Assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 565–580, 2019.

[40] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and Optimization using Miniature Simulations. In *28th USENIX Annual Technical Conference (USENIX ATC'17)*, pages 487–498, 2017.

[41] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *13rd USENIX Conference on File and Storage Technologies (FAST'15)*, pages 95–110, 2015.

[42] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 267–281, 2020.

[43] Hancheng Wang, Haipeng Dai, Meng Li, Jun Yu, Rong Gu, Jiaqi Zheng, and Guihai Chen. Bamboo Filters: Make Resizing Smooth. In *38th IEEE International Conference on Data Engineering (ICDE'22)*, pages 979–991, 2022.

[44] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems (TODS'90)*, 15(2):208–229, 1990.

[45] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing Storage Workloads with Counter Stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 335–349, 2014.

[46] 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, 2021.

[47] Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB, 2020.

[48] Juncheng Yang, Yao Yue, and KV Rashmi. A large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 191–208, 2020.

[49] 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 *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12), San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.

[50] Fan Zhang, Hanhua Chen, Hai Jin, and Pedro Reviriego. The Logarithmic Dynamic Cuckoo Filter. In *37th IEEE International Conference on Data Engineering (ICDE'21)*, pages 948–959, 2021.

[51] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *31st USENIX Annual Technical Conference (USENIX ATC'20)*, pages 785–798, 2020.

[52] Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Xiaoming Li. Low Cost Working Set Size Tracking. In *22nd USENIX Annual Technical Conference (USENIX ATC'11)*, pages 223–229, 2011.

[53] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. A Community Cache with Complete Information. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 323–340, 2021.

Technical Appendix

A Artifact Appendix

Abstract

Cuki is implemented on Alluxio. It also relies on Presto, Hive, and HDFS to function properly. We prepare the programs, assemble a workflow of Cuki and package the artifact into the Git repository.

Scope

The artifact estimates the WSS of different traces. It verifies the basic function of Cuki and validates the accuracy improvement brought by item-wise fine-grained tracking. In addition, this artifact also validates the MRC generation accuracy of Cuki is higher than the SOTA algorithm.

Contents

The artifact includes the source code of Cuki and experiments scripts. A "README.md" file can be also found in the artifact. It contains detailed description of the artifact and a step-by-step instruction for evaluation.

Hosting

The artifact is available at GitHub[1]. All branches are needed to be cloned or downloaded for evaluation. The commit version is the latest one.

Requirements

The environment of the artifact includes Hive 3.1.3, Maven 3.5.4, Hadoop 3.3.1, Java 8, Prometheus 2.37.0, Mysql 8.0.3, and S3.

B Theoretical Proof of Cuki

We first analyze the false positive rate of Cuki. Then, we theoretically demonstrate that Cuki outperforms the competitive state-of-the-art algorithms in space usage under the same false positive rate. We summarize the notations in Table 1.

Theorem B.1. *For Cuki with f-bits fingerprint and s-bits clock, the false positive rate is given by*

$$\varepsilon = 1-\left(1-\frac{1}{2^f}\right)^{2b\cdot\frac{2^s}{2^s-1}\cdot\frac{\mathcal{D}}{n\cdot b}} \approx \frac{2^s}{2^s-1}\cdot\frac{2\mathcal{D}}{n\cdot 2^f}, \tag{5}$$

[1]Our artifact: https://github.com/shadowcache/Cuki-Artifact-WSS-Estimation.

Table 1: Notations

Notations	Definition
f	Bits length of the fingerprint
s	Bits length of the clock
ε	False positive rate
b	Number of entries in a bucket
$\mathcal{D}$	Number of distinct items in a sliding window
n	Number of buckets in a Cuki
α	Load factor of a Cuki
N	Number of items in a Cuki
$\mathcal{T}$	Size of a sliding window

where b represents the number of entries in each bucket, $\mathcal{D}$ represents the number of distinct items in each sliding window, and n represents the number of buckets in Cuki.

Proof: The false positive rate of Cuki comes from two aspects: (i) Cuki stores fingerprints instead of original item keys. (ii) The outdated items in Cuki might not be cleaned up timely. For Cuki with n buckets, we define the load factor as

$$\alpha = \frac{N}{n\cdot b}, \tag{6}$$

where N represents the number of fingerprints stored in Cuki, and b represents the number of entries in each bucket.

When querying an element that does not exist in Cuki, $2\cdot b\cdot\alpha$ fingerprints need to be checked. For Cuki with f-bits per fingerprint, each check may match a wrong fingerprint and return a false positive with a probability of $1/2^f$. Therefore, the false positive rate caused by storing fingerprints is

$$\varepsilon = 1-(1-1/2^f)^{2b\alpha}. \tag{7}$$

For any item in Cuki, it will be cleaned up after performing 2^s rounds of the aging operation. For a sliding window of size $\mathcal{T}$, to prevent an item from being mistakenly deleted before its time window ends, the frequency of the aging operation is $\frac{\mathcal{T}}{2^s-1}$. Thus, for an item in the data stream, the time interval between insertion and clean-up is $\frac{2^s}{2^s-1}\mathcal{T}$. In other words, Cuki actually stores all the items inserted within the time interval $\frac{2^s}{2^s-1}\mathcal{T}$, which is $\frac{2^s}{2^s-1}$ times of the sliding window size. Suppose the number of distinct items within each sliding window is $\mathcal{D}$, the number of items stored in Cuki is given by

$$N = \frac{2^s}{2^s-1}\mathcal{D}. \tag{8}$$

Combining Equations (6), (7), and (8), we have

$$\varepsilon = 1-\left(1-\frac{1}{2^f}\right)^{2b\cdot\frac{2^s}{2^s-1}\cdot\frac{\mathcal{D}}{nb}} \approx \frac{2^s}{2^s-1}\cdot\frac{2\mathcal{D}}{n\cdot 2^f}. \qquad \square$$

Experimental verification: We conduct experiments to validate Theorem B.1. We vary f from 4 to 11, and set s as 12 - f. Other parameters follow the settings in § 6. As shown in Figure 14, the experimental results show that the theoretical false positive rate well matches the experimental results.

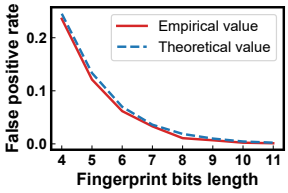


Figure 14: Verification of Theorem B.1.

Corollary B.1.1. *For a fixed memory consumption M, when $s = 1$, the minimum false positive rate is given by*

$$\frac{8\mathcal{D}}{n \cdot 2^{M/n \cdot b}},$$

where $M = n \cdot b(f+s)$ represents the memory consumption of fingerprints and clocks in Cuki, n represents the number of buckets in Cuki, b represents the number of entries in each bucket, f represents the bits length of the fingerprint, s represents the bits length of the clock, and $\mathcal{D}$ represents the number of distinct items in each sliding window.

Proof: As per Theorem B.1, the false positive rate is mainly affected by f and s. Thus we only analyze the memory consumption of fingerprints and clocks. Plug $f = \frac{M}{nb} - s$ into Equation (5), and we get

$$\varepsilon(s) = \frac{2\mathcal{D}}{n \cdot 2^{M/nb}} \cdot \frac{4^s}{2^s - 1},$$

where $s = 1, 2, \ldots, \frac{M}{nb} - 1$. Obviously, the false positive rate increases as s increases, and $\varepsilon(s)$ is the minimum when $s = 1$. This completes the proof. □

Corollary B.1.2. *For the same false positive rate ε, Cuki requires less space than ClockSketch [11] and SWAMP [5].*

Proof: According to Corollary B.1.1, let $\mathcal{T} = n \cdot b$, $n > 8$, the memory consumption of Cuki can be computed as

$$M(\varepsilon) = \mathcal{T}\log_2\frac{8\mathcal{D}}{n\varepsilon} < \mathcal{T}\log_2\frac{\mathcal{D}}{\varepsilon} \leq \mathcal{T}\log_2\frac{\mathcal{T}}{\varepsilon}. \tag{9}$$

According to [11], by ignoring the memory consumption caused by storing the size field and the payload field, the memory consumption of SWAMP is

$$M_1(\varepsilon) > \mathcal{T}\log_2\frac{\mathcal{T}}{\varepsilon}. \tag{10}$$

Therefore, to achieve the same false positive rate ε, the memory consumption of Cuki is always lower than that of SWAMP.

As per [11], the memory consumption of ClockSketch is

$$M_2(\varepsilon) \approx \frac{8}{3\ln 2}\mathcal{T}\log_2\frac{1}{\varepsilon} \approx 3.8472\mathcal{T}\log_2\frac{1}{\varepsilon}. \tag{11}$$

Let $\mathcal{T} = 2\mathcal{D}$, the memory consumption of Cuki is given as

$$M(\varepsilon) = 4\mathcal{T} + \mathcal{T}\log_2\frac{1}{\varepsilon}. \tag{12}$$

When $\varepsilon < 37.76\%$, which is often satisfied in real-world applications [17], $M(\varepsilon) < M_2(\varepsilon)$. This completes the proof. □

C Evaluation

C.1 Motivated Example of Opportunistic Aging: Estimation Fluctuation

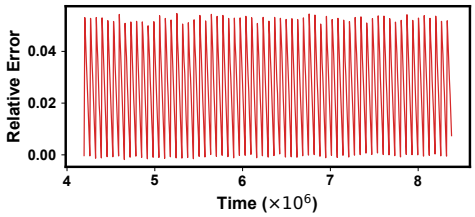
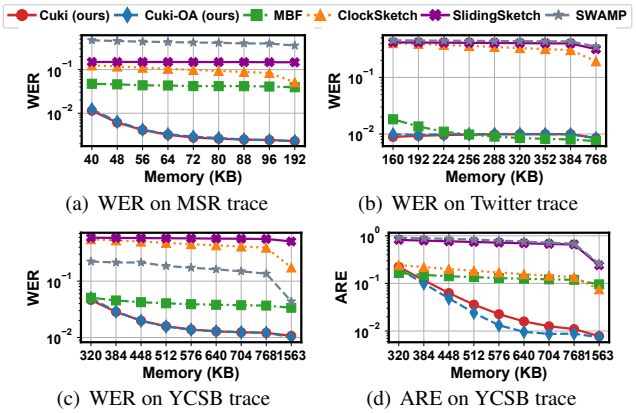


Figure 15: An example of estimation result fluctuating on the YCSB dataset (The size of a count-based sliding window is 65,536, and the clock bits is set to 4).

A large number of items will be cleared at the same time in the background aging process. As shown in Figure 15, the working set size is overestimated before aging. After the execution of aging, a tremendous amount of items are instantly cleared. Therefore the estimation result are fluctuating, and may affect the error of the estimated WSS. We propose an optimization method named opportunistic aging to alleviate this problem in aging operation.

C.2 Accuracy Evaluation of Cuki

In this experiment, we evaluate the accuracy of different WSS estimation methods. This experiment observes an additional metric WER on three traces (including the YCSB trace not shown in § 6.3), which can be seen as a supplement to § 6.3.



(a) WER on MSR trace (b) WER on Twitter trace

(c) WER on YCSB trace (d) ARE on YCSB trace

Figure 16: Performance comparison of accuracy.

Figure 16 shows the ARE and WER of different methods measured in the same run on three traces. The ARE or WER of All methods is high without sufficient memory. The ARE or WER of Cuki decreases to 1% and lower as the memory space gradually becomes larger. However, even with sufficient memory, the ARE or WER of other methods can hardly further decrease. Take the ClockSketch as an example, The WER of

ClockSketch is decreased from 12.52% to 5.13% on MSR trace as the memory increases to 1563KB. In contrast, Cuki decreases the WER from 1.14% to 0.24% as the memory increases to 768KB. This is due to the fine-grained per-item tracking method in Cuki. Although the WER of MBF is close to Cuki on the Twitter trace, Cuki performs much better in other traces. This is because MBF switches a Bloom filter out periodically and causes errors for the estimated result.

To conclude, similar to the experiment results in § 6.3, Cuki and Cuki-OA still achieve the best accuracy with the same memory consumption regarding the WER metric and YCSB trace.

C.3 Stability Evaluation of Cuki

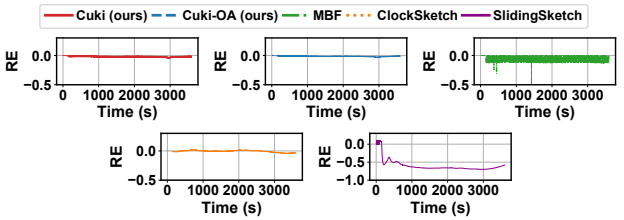
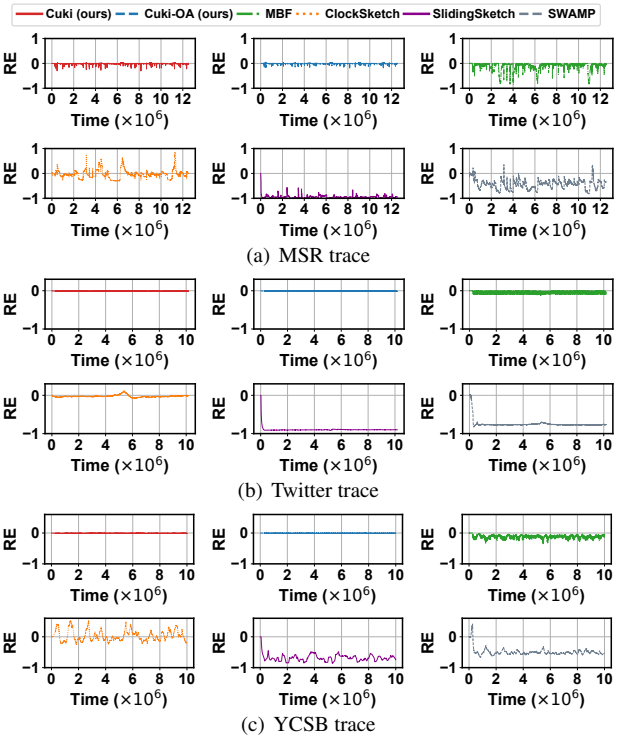


Figure 17: Performance comparison of stability in Twitter trace (time-based)



(a) MSR trace

(b) Twitter trace

(c) YCSB trace

Figure 18: Performance comparison of stability (count-based)

In this experiment, we evaluate the stability of different methods under the time-based sliding window and the count-based sliding window. For the count-based sliding window, We use the default configuration described in § 6.1. For the time-based sliding window on Twitter trace, we allocate 1408KB memory for Cuki and double memory for other methods to meet their memory requirements. We replay the Twitter trace with 24× speedup according to the data request traffic. Figures 17 and 18 illustrate the stability performance of different methods over the time-based sliding window and the count-based sliding window, respectively. The estimation results of a count-based sliding window are more stable than that of a time-based sliding window. This is because the number of items in a count-based window is fixed. However, there are still some jagged fluctuations in all methods. The reasons for these fluctuations are the same as we show in § 6.4. Benefiting from the per-item size tracking, the RE of Cuki and Cuki-OA is the most stable of the other four methods. Cuki-OA has a more stable estimation result than Cuki because of the opportunistic aging. To sum up, we conclude that Cuki and Cuki-OA also achieve the most stable and accurate estimates on a count-based window.

SAGE: Software-based Attestation for GPU Execution

Andrei Ivanov
ETH Zürich

Benjamin Rothenberger
ETH Zürich

Arnaud Dethise
KAUST

Marco Canini
KAUST

Torsten Hoefler
ETH Zürich

Adrian Perrig
ETH Zürich

Abstract

With the application of machine learning to security-critical and sensitive domains, there is a growing need for integrity and privacy in computation using accelerators, such as GPUs. Unfortunately, the support for trusted execution on GPUs is currently very limited – trusted execution on accelerators is particularly challenging since the attestation mechanism should not reduce performance.

Although hardware support for trusted execution on GPUs is emerging, we study purely software-based approaches for trusted GPU execution. A software-only approach offers distinct advantages: (1) complement hardware-based approaches, enhancing security especially when vulnerabilities in the hardware implementation degrade security, (2) operate on GPUs without hardware support for trusted execution, and (3) achieve security without reliance on secrets embedded in the hardware, which can be extracted as history has shown.

In this work, we present SAGE, a software-based attestation mechanism for GPU execution. SAGE enables secure code execution on NVIDIA GPUs of the Ampere architecture (A100), providing properties of code integrity and secrecy, computation integrity, as well as data integrity and secrecy – all in the presence of malicious code running on the GPU and CPU. Our evaluation demonstrates that SAGE is already practical today for executing code in a trustworthy way on GPUs without specific hardware support.

1 Introduction

Fueled by recent trends such as machine learning and the declining yields from Moore's Law, the use of accelerators to process the vast volumes of data is becoming indispensable. In fact, it is expected that the majority of compute cycles in public clouds will be executed on accelerators [29].

With the application of machine learning to security-critical or sensitive domains such as healthcare or financial modeling, there is a growing need for a mechanism that maintains integrity and secrecy for both code and data despite the computation being offloaded to the GPU.

With the wide-spread deployment of trusted execution environments (TEEs), e.g., Intel SGX [4] and ARM TrustZone [2], an important question is how security-sensitive computation tasks can be accomplished on GPUs. While first hardware-based TEEs on GPUs are starting to emerge [13, 20, 24, 27, 46, 49], how can we execute code securely on GPUs *in current environments*? As we have witnessed from the introduction of hardware-based TEEs on x86 platforms, it took over a decade until it became possible to fully and widely utilize these mechanisms. At the same time, technology progress in this space is a moving target as new attacks (among other factors) force vendors to phase out one specific hardware-based technology in favor of more robust successors (such as with the case of the deprecation of Intel SGX [39]). Given the importance of software executing on GPUs, it is clear that we need to find approaches to speed up the long lag time between deployment and wide-spread utilization.

A promising approach for bridging this gap is a software-only approach to trusted execution. In the context of CPU-based execution, a rich research field has contributed numerous approaches [8, 32, 33, 48]. The basic idea of the prior software-based or timing-based attestation approaches was to design a verification function that would run on an untrusted system and compute a checksum over itself – where both the correctness of the checksum and the time duration are measured by a trusted verifier. A correct checksum value that is returned before a threshold point in time, indicated to the verifier that the TEE was correctly set up and that the correct code is now executing (code integrity and launch point integrity). In combination with a system for control-flow verification, control-flow integrity can also be achieved.

The challenge of such software-based TEE establishment approaches lies in the creation of a verification function that will slow down noticeably or produce an incorrect checksum, if an adversary attempts to tamper with its execution.

The creation of a verification function for GPU environments poses numerous research challenges, which may be the reason why it has so far not been achieved, to the best of our knowledge. First and foremost, achieving (1) code se-

crecy and integrity, and (2) data secrecy and integrity, (3) in the presence of a malicious OS, (4) malicious code on GPU, and (5) a malicious CPU-GPU interconnect is a formidable challenge. Other challenges that we have to overcome include the absence of a true random number generator on the GPU, the lack of documentation from GPU vendors for a specific target architecture, no toolchain support to write native GPU microcode, and the difficulty in achieving optimal GPU utilization.

We design the SAGE system, which establishes a TEE on NVIDIA GPUs of the Ampere architecture (A100). SAGE utilizes an SGX enclave running on the host to act as a local verifier, and to bootstrap the software primitive to establish a dynamic root-of-trust (RoT) on the GPU. RoT establishment ensures either that the state of an untrusted system contains all and only content chosen by a trusted local verifier and the system code begins execution in that state, or that the verifier discovers the existence of unaccounted content. SAGE also sets up a shared secret key between the verifier and the GPU, which can be used to establish a secure channel to achieve integrity and secrecy for code and data transferred. Our results indicate that after a successful invocation of SAGE, the verifier obtains assurance that: (1) the user kernel on the untrusted device is unmodified; (2) the user kernel is invoked for execution on the untrusted GPU device; and (3) the user kernel is executed untampered, despite the potential presence of a malicious actor.

This paper presents the following contributions:

- We design a software-based attestation mechanism for GPU execution that enables secure code execution on NVIDIA Ampere GPUs, providing code integrity and secrecy, computation integrity, as well as data integrity and secrecy.
- We implement the race-condition TRNG and Verification Function used as basic security components in the software TEE. This requires an understanding of the GPU architecture and the format of the instructions used in the microcode, which we derive from our decoding and instruction generation framework.
- Through a proof-of-concept implementation on the NVIDIA A100 platform, we demonstrate the technical feasibility of the approach. Our implementation is publicly accessible at `https://github.com/spcl/sage`.

2 Background: GPU Fundamentals

In the following, we describe the fundamentals of NVIDIA GPUs and their programming model (CUDA) to illustrate how compute tasks are offloaded and executed on the GPU.

The GPU is connected via the PCI control engine to the host CPU and uses an internal bus for communication between its core components. The core components are the command processor, compute and DMA engines, and the memory system, consisting of a memory controller, registers, on-chip and device memory.

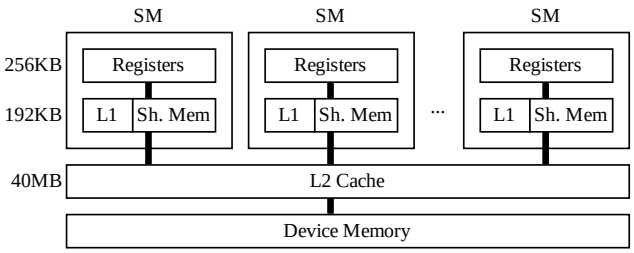


Figure 1: Memory hierarchy of a GPU with memory sizes of NVIDIA A100 GPU.

Controlling the GPU. Commands to the GPU are transmitted using a set of command queues known as *channels*. The GPU's command processor receives these commands and forwards them to the corresponding engines.

Data transfer to the GPU. GPU programming inevitably incurs data transfers between host and device memory. This is handled using direct memory access (DMA). The copy engine is responsible for handling DMA commands and their corresponding memory accesses.

GPU execution. The GPU's compute engine contains multiple Processor Clusters (PCs), each containing multiple Streaming Multiprocessors (SMs). SMs are partitioned into multiple processing blocks, each containing specialized processing cores (e.g., INT32 cores), a scheduler and a dispatch unit. *GPU kernels* to be executed on the GPU are scheduled to SMs and specify the number of threads to be created. These threads are organised in thread blocks and grids. Thread blocks are divided into warps. Each warp is a group of 32 parallel threads and gets scheduled by a warp scheduler.

Modern GPUs have multiple processing pipelines [23] for different data types. The FMA pipeline executes 32-bit floating point instructions and integer multiply and add (`IMAD`). The ALU pipeline executes 32-bit integer, logical, binary, and data movement operations. In addition, there are pipelines for 64-bit and 16-bit floating point, and Tensor core operations.

GPU memory system. The memory system on GPUs consists of a memory controller and different memory levels. The memory levels are associated to the compute system as follows (see Figure 1). Each processing block includes an L0 instruction cache and a register file. The combined processing blocks of a SM share a combined L1 data cache/shared memory that can be partitioned depending on the workload. Multiple SMs share an L2 cache before pulling data from global (off-chip) GDDR memory. *Registers* are a shared resource and are allocated among the thread blocks executing on a SM. Accessing a register consumes zero extra clock cycles per instruction, but delays may occur due to register read-after-write dependencies and register memory bank conflicts. In case a thread requires more registers than available, the data contained in the registers is spilled into shared memory. *Shared memory* is not only used for register spilling, but also enables communication and memory reuse between threads in a block.

3 Problem Definition

We first describe the design goals we strive to achieve, as well as the assumptions and the adversary model we consider.

3.1 Design Goals

Verifiable code execution on the GPU. Verifiable code execution describes the problem in which a verifier wants a guarantee that some arbitrary code has executed untampered on an untrusted platform, despite the potential presence of a malicious entity (e.g., malicious software) [32]. This problem is typically approached by verifying code integrity through root of trust attestation, setting up an untampered code execution environment, and then executing the code.

Data integrity and confidentiality. In addition to code integrity also the integrity and/or confidentiality of the data executed on the GPU must be ensured. Specifically, we aim to guarantee that the adversary cannot observe or tamper with data transferred to/from the GPU by a trusted application that runs in a CPU TEE.

Dynamic root of trust without hardware support. Dynamic root of trust establishment denotes the problem of dynamically setting up a trusted computing base (TCB) on an untrusted platform without hardware support. All code contained in the dynamic root of trust is guaranteed to be unmodified and it can thus be used to provide externally verifiable code execution.

3.2 Assumptions

Verifier and GPU on the same machine. We assume that the verifier is executed on the same machine as the GPU we want to attest. The GPU is directly connected to the host CPU over a bus (e.g., PCIe with a latency of ~500 ns [18]).

GPU hardware configuration. We assume that the verifier knows the exact hardware configuration of the GPU, including the GPU model, the number of cores, the memory architecture, and the GPU clock speed. This assumption is practical when the hardware configuration is managed by the user or a trusted cloud provider. The machine owner has knowledge of the hardware configuration, which cannot be altered by software. In this configuration, we aim to protect against remote attackers who may arbitrarily modify software.

3.3 Threat Model

In the following, we discuss the threat model by defining the trusted compute base (TCB) and outlining the capabilities of an adversary. The TCB of a system refers to all hardware and software components that are critical to its security, in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system.

Trusted compute base (TCB). We assume that the remote adversary has full control over the software of the untrusted host system. In other words, the adversary has administrative privileges, can tamper with the operating system, or the guest operating system and the hypervisor in case of virtualization. However, we assume that the hardware primitives of the CPU and GPU, including firmware are contained in the TCB. Since SAGE uses Intel SGX, it inherits the TCB of SGX (which includes the CPU package, trusted libraries, etc.).

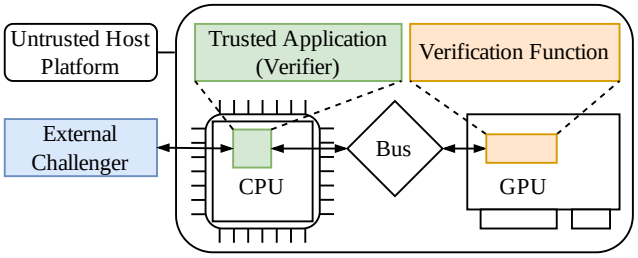


Figure 2: Abstract system model.

Capabilities. Considering these capabilities, an adversary can read and tamper with code or data of any victim process, and can access or modify data in DMA buffers or commands submitted to the GPU. Furthermore, the adversary could inject packets in arbitrary locations on the I/O communication path between the host and the GPU. This gives the adversary control over attributes, such as the address of GPU kernels being executed and parameters passed to the kernels. The adversary may also access device memory directly over MMIO, or map a user's GPU context memory space to a channel controlled by the adversary. In GPUs that support multi-tasking, malicious kernels can be dispatched to the GPU, thereby accessing memory belonging to a victim's GPU context.

Out of scope. Since this work tackles the problem of trusted execution on the GPU, we do not consider attacks that target SGX, such as physical attacks to the CPU package or side-channel attacks on SGX. In addition, we do not consider system availability attacks that prevent the execution of our process, as an adversary with the described capabilities can always prevent the deployment of computing tasks on the GPU. We assume that the adversary is not capable of using undocumented GPU capabilities to execute an attack. We believe that it is the responsibility of the manufacturer to ensure that such attacks are not possible, since the details of hardware and driver implementations are hidden from users.

4 SAGE Overview

SAGE addresses the problem of verifiable code execution on a GPU without hardware support, in which the verifier wants a guarantee that the user kernel has executed untampered on an untrusted GPU platform, even in the presence of an adversary. Figure 2 illustrates the abstract system model we consider.

SAGE comprises two main components. The first component is the verifier, which runs as a trusted application on the host CPU (e.g., using Intel SGX [4]) and is attested by an external challenger. The second component is the verification function (VF), which runs on the the untrusted GPU. The VF computes a checksum over its own code, and is constructed in an intricate way such that if a change is applied to the VF then either the execution will slow down in an externally

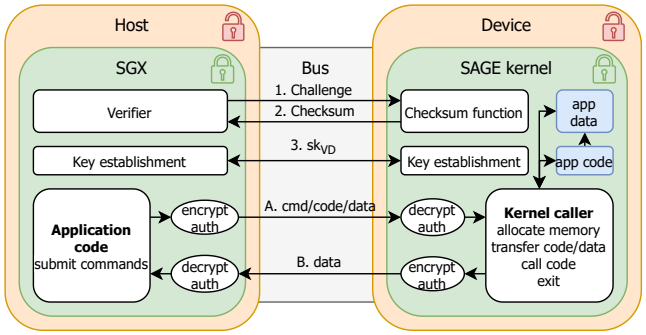


Figure 3: Overview of SAGE. The numbers represent temporal ordering of events. The letters show repetitive operations.

detectable manner, or the checksum value will be incorrect.

The verifier dispatches to the GPU the VF and then invokes it with a challenge while measuring the VF execution time. The VF computes a checksum value and returns it to the verifier. Using the same VF logic, the verifier independently computes and verifies the correctness of the checksum value. If the checksum returned by the VF is correct *and* it is returned within the expected time, the verifier obtains a guarantee that a dynamic root of trust on the GPU was established.

Once the dynamic root of trust has been established, the VF sets up an untampered execution environment. During the setup of the execution environment, a shared key between the verifier enclave and GPU is established; afterwards, only commands authenticated with this key are accepted, including the movement of encrypted kernel code and data between host and GPU. SAGE guarantees execution integrity and memory protection for the code and data stored in the GPU memory (see §8). Figure 3 shows an overview of SAGE including a sequence of events.

5 Verification Function (VF)

The VF that runs on the untrusted GPU is the fundamental component of SAGE. We now describe in detail these tasks and the challenges they entail.

5.1 Design Requirements

The VF must be carefully constructed in such a way that if an adversary were to tamper with the VF or the user kernel, it would result in either a wrong checksum or a noticeable time delay. Before offering a concrete design for the VF, we describe several required properties and outline how these properties influence the correctness of the checksum or the VF execution time. We defer our security analysis to §8; the following properties also account for the attack surface analyzed therein.

Time-optimal implementation. The implementation of the VF must be *time-optimal*. Otherwise, the adversary could use a faster implementation and use the time saved to forge the checksum (e.g., by injecting instructions).

Maximize resource usage during checksum computation. To prevent the adversary from running any other computation during the checksum computation, the VF maximizes its resource usage on the GPU by using all available SMs and avoiding "empty" threads. Moreover, each thread should use the maximum number of available registers to prevent the adversary from using those registers. Thus, if a malicious computation attempts to use more registers than available, the values of the affected registers are spilled into shared memory, resulting in a noticeable execution time difference (4- vs. 30-cycle latency for registers and shared memory, resp.).

Predictable execution time. The execution on GPUs is optimized to achieve high data throughput with deterministic latency, but the execution time is non-deterministic (e.g., due to multi-threaded execution, scheduling, and caching). The VF execution time should have low variance so that the verifier can predictably determine the execution time.

Challenge-dependent checksums. To prevent the adversary from pre-computing the checksum before making changes to the VF, and to prevent the replay of old checksum values, the checksum needs to depend on an unpredictable challenge sent by the verifier.

5.2 Concrete VF Design

The VF consists of initialization, self-verifying checksum function to establish a dynamic root-of-trust, and establishing an untampered execution environment including a key establishment protocol between the verifier and the GPU.

During the initialization phase, the memory buffer is allocated on the GPU and returned to the verifier. Then the VF code is copied into the buffer.

5.2.1 Self-Verifying Checksum Function

The checksum function is used to obtain a guarantee that the integrity of the VF code running on the GPU is unaffected by an adversary. For this purpose, the checksum function computes a checksum over the entire VF code. The resulting checksum can be used as a *fingerprint* of the VF and enables detection of changes to the VF code. If an adversary modifies the VF code, the checksum will differ with high probability. Thus, once the verifier receives a correct checksum within a threshold time, it has a guarantee that the VF code running on the GPU is unmodified.

Since the checksum computation code is part of the VF and will thus be included in the checksum calculation, the checksum function computes the checksum over its own instruction sequence and verifies itself. This property is further referred to as *self-verification*.

Checksum initialization. GPUs contain multiple multiprocessors that can be used for parallel execution. To achieve the maximal computational power of a GPU, the verifier sends a set of challenges containing a specific challenge value for each multiprocessor. Upon receiving a set of challenges, each multiprocessor uses its challenge as a seed value to initialize all per-thread state with pseudo-random data. Each thread has its own set of registers which are used to store the run-

ning checksum values and a data pointer. The data pointer references the VF code in the initially allocated buffer.

Checksum loop. The checksum computation is performed iteratively. Each iteration executes the same number and type of instructions and has a constant execution time.

Pseudo-random memory access prevents the adversary from predicting which instruction will read the potentially-modified memory location and forces the adversary to monitor every memory read by the checksum code, resulting in a noticeable time overhead. Indirectly, this process performs the inclusion of the data pointer in the checksum to prevent memory copy attacks (see §8).

Update the checksum. The running checksum values are updated to include the accessed VF code into the checksum value using a sequence of instructions. To achieve a time-optimal implementation, we use simple arithmetic and logical instructions (e.g., +, <<, >>, etc.) that are challenging to implement faster or with fewer operations. Taking inspiration from the strong ordering in [32], the instructions used to update the checksum alternate between arithmetic and logical instructions to enforce a strong ordering of the instructions.

Self-modifying code. The instructions of the self-modifying code fragment depend on current value of the checksum and are changed in each iteration of the checksum function. In our case the current value of the checksum function is used as an immediate value for an instruction (see §6.5 for details).

Checksum epilogue. Since the checksum computation is conducted using individual threads located on different multiprocessors, the checksum values need to be aggregated before sending the checksum result back to the verifier. This aggregation is conducted in three steps. First, we aggregate the checksum per warp. Each of the per-thread checksums is added pairwise to obtain a warp-level checksum. Second, the warp-level checksums are aggregated by thread block using shared memory. Finally, we aggregate the checksum per grid using global memory. Each of the aggregation steps uses a pairwise addition (which is mapped to an atomic add instruction in native assembly). The final result of the checksum computation is then sent to the verifier.

5.2.2 Untampered Execution Environment

After establishing a dynamic root-of-trust on the device, the VF sets up an execution environment in which the user kernel is guaranteed to run untampered. This includes setting up a shared secret between the verifier and the device, and checking the authenticity of the user kernel to be executed on the GPU using a hash function. The shared secret can then be used to authenticate and encrypt commands and data sent by the verifier to the device and vice versa.

Key establishment. To establish a shared secret between the verifier and the device, we rely on the SAKE protocol [31], a protocol for key establishment between neighboring nodes in sensor networks without requiring any prior secrets. The protocol is based on the Diffie-Hellman key exchange protocol and uses the Guy Fawkes protocol [1] for authentication. The Guy Fawkes protocol is based on hash chains and relies on the property that each of the participants needs to authenticate the other party's hash chain. In SAKE, this authentication is achieved using software-based attestation and exploits the asymmetry in the computing time between the genuine checksum function executing on the device and an external entity computing the checksum value. This allows us to use the resulting checksum as a short-lived secret. Furthermore, the SAKE protocol assumes that the adversary does not introduce any computationally more powerful nodes into the network, which aligns with the assumptions for SAGE (see §3.2).

To apply the SAKE protocol to SAGE, we change the protocol as follows: 1) The checksum function in SAKE that was proposed for the use in sensor networks is replaced with SAGE's checksum function. 2) Instead of both participants acting as challengers, only the host enclave will engage as a challenger. 3) We replace the cryptographic primitives used in the protocol with AES-CMAC as the MAC function and SHA256 as the hash function.

The key establishment protocol in SAGE works as follows. First, the verifier sets up its own hash chain for the Guy Fawkes protocols and DH public key as:

$$V: \quad v_0 = g^a \bmod p \quad v_1 = H(v_0) \quad v_2 = H(v_1) \tag{1}$$

where a is a random bitstring $a \leftarrow^{\mathcal{R}} \{0,1\}^n$ Then, it sends v_2 to the device and records the current time as t_0.

$$[t_0] \quad V \rightarrow D : v_2 \tag{2}$$

Upon receiving v_2, the device uses it as a challenge for the checksum function and then uses the computed checksum and a random value to generate its own hash chain and replies to the verifier:

$$D: \quad w_0 = H(c \,\|\, r) \quad w_1 = H(w_0) \quad w_2 = H(w_1) \tag{3}$$

where r is a random bitstring $r \leftarrow^{\mathcal{R}} \{0,1\}^n$, c is the result of the checksum computation and $\|$ denotes to concatenation.

$$[t_1] \quad D \rightarrow V : w_2, \ \mathrm{MAC}_c(w_2) \tag{4}$$

The verifier checks if the measured execution time $(t_1 - t_0)$ matches the expected execution time and aborts the protocol otherwise. In the meantime, the device sets up its own DH public key:

$$D: \quad b \leftarrow^{\mathcal{R}} \{0,1\}^n \quad k = g^b \bmod p \tag{5}$$

Then, the verifier and the device gradually disclose the remaining of their hash chains to each other:

$$V \rightarrow D : v_1 \qquad D \rightarrow V : w_1, \ k, \ \mathrm{MAC}_{w_0}(k) \tag{6}$$

$$V \rightarrow D : v_0 \qquad D \rightarrow V : r \tag{7}$$

For each message the recipient checks whether the received value matches the expected hash chain. Finally, the verifier V and the device D compute the shared secret key sk_{VD}:

$$sk_{VD} = k^a = (g^b)^a \bmod p \quad sk_{VD} = v_0^b = (g^a)^b \bmod p \tag{8}$$

After the dynamic RoT has been established on the GPU and the integrity of the user kernel has been checked, the host enclave can start transferring code and data to the GPU. Depending on the sensitivity and security criticality of the domains, the data could be either *authenticated* and/or *encrypted* using the established symmetric key sk_{VD}.

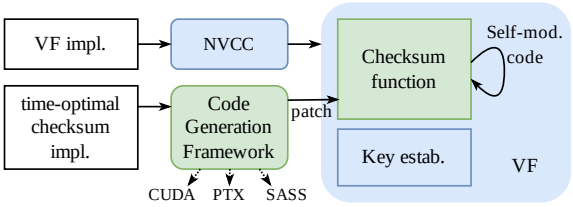


Figure 4: Code pipeline to generate the VF microcode. The green blocks generated using our framework.

6 Implementation

The requirements to achieve a time-optimal (see §5.1) implementation on the Ampere architecture (further discussed in §6.3) include maximizing GPU utilization, consuming all available compute resources, optimally filling the processing pipelines, and optimizing cache usage.

Unlike the higher levels of the CUDA computing platform such as the CUDA C++ language extension and the parallel thread execution (PTX) virtual machine and instruction set architecture, NVIDIA provides very little information about the hardware-specific instruction sets for a specific target architecture. Moreover, even if one resorts to write inline PTX virtual assembly, the Streaming (or Shader) Assembler (SASS) code emitted by the compiler often does not achieve the performance of native GPU applications. The execution of microcode that has been compiled using the regular CUDA compiler often is on the order of 10x slower compared to optimized microcode [14, 15]. As a consequence, libraries used for high-performance computing (e.g., cuBLAS [25]) contain highly optimized microcode tailored to a specific architecture. In addition to the performance gap to native GPU code, the user has no control over the translation from PTX virtual assembly to the SASS assembly for the target architecture.

To achieve a time-optimal implementation, we needed to implement a custom instruction generation framework that allows patching of binary microcode with a highly optimized version. The implementation of this framework requires understanding the Ampere architecture and the instruction format used in microcode. Although our focus in this paper is on the A100, we expect that small modifications to the code generator can provide support for the Volta and Turing architectures as well. Figure 4 illustrates the pipeline used to generate the VF. The VF is implemented using CUDA C++ and compiled using `NVCC`. However, the section containing the checksum function is patched using an optimized implementation generated as binary microcode using our framework.

6.1 Instruction Decoding

To understand the instruction format used in the recent Ampere GPU architectures, we implemented a framework that allows decoding of instructions using `cuobjdump` and `nvdisasm` [21] by decoding handcrafted code samples and samples from existing CUDA libraries (e.g., cuBLAS [25]).

Instruction format. NVIDIA's Ampere architecture adopts the same general instruction format as its predecessors Turing and Volta [14, 15]. All these architectures use 128 bits to encode both an instruction and its associated scheduling control information. The encoding that is used in these architectures is fixed length and uses similar encodings for all instructions. Figure 5 illustrates a typical instruction encoding.

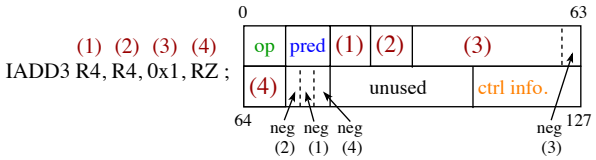


Figure 5: Instruction as decoded by `nvdisasm` and its format. `pred` denotes predicates, `op` refers to the operation code, and `neg` allows negating the corresponding parameter.

Control information. The control information section in the instruction encodes scheduling decisions taken by the compiler that the hardware must enforce. The control information is organized as follows: reuse flags (4 b), wait barrier mask (6 b), read barrier index (3 b), write barrier index (3 b), yield flag (1 b), and the number of stall cycles (4 b). The reuse flags allow data reuse between instructions without accessing any register ports. The wait barrier mask and indices are used for instructions with variable latency (e.g., instructions involving a memory access). These dependency barriers can be used to enforce the completion of variable-latency instructions. The yield flag is used to balance the workload assigned to a processing block. The stall cycles indicate the latency of the instruction before issuing the next instruction. Jia et al. present a detailed description of the control information [14].

6.2 Instruction Generation

Understanding the instruction format allows us to generate the specific instructions we need for our implementation. These instructions then need to be translated to the correct binary format. For this purpose, we implement an instruction generation framework that allows emitting instructions either in CUDA C++, the virtual assembly language PTX, or as binary microcode that is natively executed on the GPU.

The instruction can be defined in the following format, where the section separated using `|` symbol describes the control information for the instruction (barrier mask B, read barrier index R, write barrier index W, yield flag Y, and number of stall cycles S):

```
B......|R.|W.|Y1|S1| IMAD.U32 R28, R28, 2048, R28;
```

Our instruction generation framework then translates the instruction to the selected target language (CUDA C++, PTX, microcode). This allows us to rapidly prototype checksum functions and compare performance between implementations in each of the languages.

6.3 Time-optimal Technical Requirements

We formulate the following technical requirements for a time-optimal implementation of the checksum function. These are subject to characteristics of the target architecture; in our case, the NVIDIA Ampere architecture.

Maximize resource consumption. To maximize the resource consumption during the checksum computation, the checksum function must use all available compute resources. The NVIDIA A100 GPU has 108 Streaming Multiprocessors (SMs) each containing 64 `FP32` and 64 `INT32` units [19] that must be used during each clock cycle.

Optimally fill FMA and ALU pipelines. Since both the FMA and ALU pipelines have an instruction issuing latency of 2 clock cycles, `FP32` and `INT32` instructions must be interleaved to fully saturate both pipelines. In addition, instructions that use registers with a direct dependency must be executed with a latency of at least 4 clock cycles to avoid pipeline stalls (e.g., read-after-write dependency).

Optimal GPU utilization. To achieve full GPU utilization, the number of threads per thread block needs to be picked according to the target architecture. The A100 achieves full GPU occupancy by assigning 2 blocks of size 1024 to all the 108 available SMs (216 total). Each SM has 65,536 32-bit registers available for threads. To use all registers during the checksum computation while maintaining full utilization of the GPU, 32 registers are assigned per thread [22].

Cache size. The code blocks should not exceed the capacity of L0 and L1 instruction caches (see Figure 1).

6.4 Selection of Optimal Overheads

An optimal implementation of a checksum function should perform a useful computation step in each clock cycle. In practice, this requires a highly optimized use of the underlying hardware. In the following, we show a recipe for building such a checksum function for the A100 GPU.

Unutilized clock cycles are mainly caused by instruction cache misses, global memory access latency, pipeline stalls, and jumps. In the beginning of each clock cycle, the SM warp scheduler selects a subset of warps (up to `S=4` on A100) from all active warps (up to `A=64` on A100) to execute. This selection mechanism can avoid performance losses if at least `S` are ready to execute on each clock cycle.

To analyze the performance of the checksum function, we use a simplified model of the number of clock cycles per instruction. In the following, we will demonstrate that it helps to reach the performance with the precisely specified number of clock cycles. We distinguish the total number of useful clock cycles `X` and overhead cycles `Y`, so that the total number of clock cycles spent by the code using a single thread is `X+Y`. For example, with proper instruction ordering to avoid pipeline stalls, an `IMAD` instruction has `X=1` and `Y=0`. An instruction reading from global memory has `X=1` and approximately `Y=250`. To prevent attacks on the checksum function by executing some instruction each clock, the value of `Y` must not exceed `X(A/S-1)`. Then, the GPU scheduler will be able to completely hide the overhead `Y` so that the actual amount of time spent will be `X`.

Integer shifts and multiplications with addition directly affect the result of the checksum calculation. However, the instruction to jump from the end of the loop body to its beginning does not change the checksum. The attacker may try to unroll a few iterations of the loop to save the clock cycles required to perform this jump (and potentially misuse them for an attack). To prevent such attacks, we unroll the loops until it is not possible to unroll them further without causing instruction cache misses. The target value `Y` for unrolling must be so large that one additional instruction cache miss will increase it to `Y′` without the possibility for a hardware scheduler to compensate for the increase (and potentially hide it) using scheduling.

In practice, we have noticed that achieving this level of control over the order of instructions, and the arrangement of unrolled loops is very difficult without vendor support: the documentation on SASS and hardware details is deliberately kept closed to reduce backward-compatibility issues. It is especially difficult to control instruction cache misses because of the use of self-modifying code to protect against memory copy attacks. The only way to invalidate the instruction cache on the A100 is to overflow it with the block of instructions of the cache size, so controlling the value of `Y` by changing the size of the checksum function is not possible. That leaves only memory accesses and jumps that can change `Y`. We assume that adding an instruction to invalidate the instruction cache requires minimal (or no) changes to the GPU architecture because a similar instruction already exists for the data cache (`discard` in PTX ISA or `CCTL` in SASS).

6.5 Implementation of SAGE

Verifier. We implement the verifier enclave using the Intel SGX SDK [11] and its `tcrypto` library [10]. The enclave creates a CUDA context on the GPU, loads the VF as a module, and calls the VF kernel. To generate nonces in the enclave that are then transferred to the GPU as challenges, we use AES-CTR with an IV that has been generated using a TRNG during the enclave creation.

VF. The VF is implemented in CUDA C++, except the checksum function component, which is patched by binary microcode using our framework. The checksum function executes a loop containing the following operations.

First, the iteration counter is increased and checked if the maximum number of iterations is reached. Then the VF data block `D` is read from memory from the location defined by the current checksum value `C`, used as an offset: `D=data_ptr+(4×C mod data_size)`. After the load is complete, it is included in the checksum `C+=D`.

The read from main memory may take 250 – 500 cycles to be completed. The GPU compiler sets a read barrier for this instruction and the GPU stalls the compute pipeline until

the read has been completed. Instead of the stall, we develop an instruction pattern that is executed while waiting for the memory read to complete ("busy waiting"). We use interleaved (see §6.3) `X+=X<<N` with `IMAD` (FMA) and `X+=X>>N` with `LEA.HI` (ALU) instructions where `X` is any of 32 registers. The security of this computation depends on the existence of an alternative sequence of instructions which can compute the result faster. We expect that for some cases of long sequences or poorly chosen shift amounts, it is possible to find a shortcut constructed similarly to the jump ahead function in the xorshift pseudo-random number generator (PRNG) [45]. To prevent such shortcuts, we partition long sequences by requiring materialization of intermediate values, breaking them with random memory accesses included in the checksum. We aim for sequences of such length that the cost of implementing a shortcut is higher than performing the actual computation.

After updating the checksum function, we compute the self-modifying code that consists of the following binary instruction: `C+=C>>N`, where the immediate `N` depends on the current checksum value. We overwrite immediate parameter with the current value of the checksum. Thus, the value of `N` changes for each iteration and ensures that we are executing the code that we are verifying. To avoid race conditions when updating the immediate value of these instructions, these instructions are required to be located in different memory areas for each thread block.

6.6 Random Number Generation on GPUs

For the key establishment protocol based on the modified SAKE protocol, the GPU needs to be able to generate random values. Given that the adversary knows the entire code executing on the GPU, we cannot use a secret provided by verifier to initialize the PRNG used in the protocol, but instead must rely on a true random number generator (TRNG).

TRNG implementation on GPUs. Approaches that use physical unclonable functions (PUFs) to initialize PRNGs on the GPU [7, 30, 43] are not practical to be used in SAGE as they either require resetting the GPU or use features that are under control of the adversary (e.g., voltage supplied to the GPU). Consequently, we use a TRNG implementation which is based on race conditions in multi-core environments caused by simultaneous memory accesses to shared variables. It takes advantage of uncertainties that arise when cores simultaneously access a particular memory location [40]. In our case, each simultaneous memory access unpredictably flips bits stored in shared variables. This unpredictability enables the GPU to generate noise which can be sampled and then used as an entropy source. We evaluated our implementation using statistical tests such as NIST SP 800-22 [36], DIEHARD [17], and ENT [47]. The TRNG implementation passes all standard tests and achieves a throughput of 4 kB/s on NVIDIA A100 GPUs and thus takes around 8 ms to generate an output of 256 bits. The TRNG provides 7.999 996 bits of entropy per byte (measured using ENT [47]).

7 Evaluation

Evaluation setup. To evaluate the performance of the checksum function, we use a setup based on an ASUS RS720-E10-RS12E equipped with a A100-PCIE-40GB GPU and Intel Xeon Gold 6348 CPU [12] which natively supports SGX instructions. We run the SGX enclave in both native and simulation mode. To benchmark the execution time of the verification process and evaluate runtime overheads, we also run the VF on a dual-socket system with an A100-SXM4-40GB and AMD EPYC 7742 CPU.

Register consumption. For the execution of the checksum function, the loop counter, data pointer, and the checksum result are stored in registers. In addition to those registers, we use 22 additional registers to store intermediate state during the computation of the checksum. In total, the checksum function verifies 524,288 bytes. The beginning of the buffer contains the checksum function itself, whereas the remainder is filled with pseudo-randomly generated values.

Experiment Nr.	1	2	3	4
self-modifying code	✗	✗	✓	✓
instructions	428	429	8,342	8,342
iterations	100,000	100,000	1,000	1,000
inner iterations	0	0	0	5000
inner instructions	0	0	0	216
verification (AMD) [s]	21.6	21.6	9.99	497
verification (Intel) [s]	102	102	47.0	2337
runtime T_{avg} [s]	0.4941	0.4977	0.1309	12.40
% of GPU peak perf.	99	98	75	100
adversarial NOP	✗	✓	✗	✗
runtime σ [s]	0.0009	—	—	—
runtime T_{min} [s]	—	0.4966	—	—
$T_{avg} + 2.5\sigma$ [s]	0.4964	—	—	—

Table 1: Evaluation of checksum implementations.

Summary of results. Table 1 summarizes our experiment series conducted to evaluate the performance of SAGE's VF. We distinguish between two categories depending on whether the checksum function contains self-modifying code or not. Depending on the category, the total number of instructions and number of checksum loop iterations are adapted. For each experiment, we report the VF's execution time on the GPU, the utilization ratio during the checksum execution, the verification time on the CPU, detection threshold, etc.

Experiment 1 demonstrates our best reference implementation. Experiment 2 simulates an attack on the checksum function from the first experiment. In Experiment 3, we show the effect on the performance of adding self-modifying code to the reference implementation. Experiment 4 shows a possible technique to compensate for the loss of performance with enabled self-modifying code.

7.1 VF Performance

To evaluate the performance of VF, we report its average runtime and utilization ratio during the checksum execution (Table 1). As a reference for this ratio, we use the *peak GPU performance*, which assumes that the number of warps that are executed concurrently per clock cycle is 4 (see §6.4).

We compare our reference implementation from Experiment 1 (in SASS) with the same code written in PTX (virtual assembly), that has been processed using the NVIDIA PTXAS assembler with the highest possible level of optimization enabled. In comparison, the optimized version of the checksum function that we generated using our instruction generation framework is around ∼230% faster than an implementation in PTX.

The checksum functions in Experiments 3 and 4 contain self-modifying code. This requires triggering cache eviction of the instruction cache such that the modified instruction gets updated. To trigger the cache eviction for the L2 instruction cache (128 kB), the checksum loop is required to be larger than the cache size. As a consequence, we use 8342 16 B instructions in the checksum loop. With this cache eviction strategy, our implementation is able to achieve 75% of the maximum utilization. Upon closer inspection with a GPU profiler, we find that 99% of all pipeline stalls that happen during the execution of the checksum function are caused by the fact that no instructions are available in the instruction cache to be executed. On average, each warp of this kernel spends 14.1 cycles being stalled due to not having the next instruction fetched yet. In comparison, reducing the size of the checksum loop to 6.7 kB (as in Experiment 1), we achieve a utilization of 99% without triggering cache eviction. This means that the hardware is unable to load the modified instructions in time for execution without causing any pipeline stalls. By comparing the VF's performance in Experiments 1 and 3, we can conclude that a higher utilization can be achieved in case other cache eviction strategies become available (see §6.4).

In addition to the previous experiment, we modified the checksum function by adding an "inner" loop to the main loop of the checksum function calculation (Experiment 4). This effectively hides the performance loss due to cache misses in the instruction cache and achieves 100% of the GPU peak performance. However, the time required to verify the code outside of the nested loop drastically increases and is thus considered too long to be practical.

7.2 Attack Robustness

To evaluate the robustness of our VF implementation with regards to attacks, we estimate the number of instructions that can be injected by an adversary without causing a noticeable time overhead. For this purpose, we measure the performance of the checksum function for 100,000 iterations and record the standard deviation σ of the total execution time based on 100 runs. We assume that the results of this experiment series are normally distributed and set the threshold value to detect adversarial tampering to be at $2.5 \cdot \sigma$ from the mean. The probability of a false positive is about 0.5%, in which case the verification process is restarted. Depending on the application requirements, the probability of false positives can be decreased at the expense of a larger number of iterations.

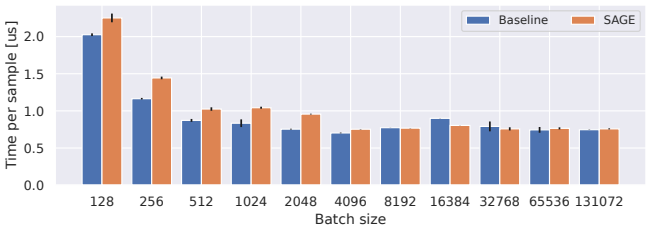


Figure 6: Time per batch sample in MLP.

To evaluate the robustness of this approach, we insert one additional NOP instruction in Experiment 2 (adversarial NOP) and report the minimum run time T_{min} (averaged over 100 runs). Assuming a detection threshold of $T_{avg} + 2.5\sigma$, we can conclude that $T_{avg} + 2.5\sigma < T_{min}$ and thus it is impossible to insert one or more instruction without detectable overhead.

Time measurement occurs on the CPU, including the time of communication between the CPU and GPU, in addition to the checksum computation loop. This raises the question of performance portability across hardware configurations. Since the configuration is fixed and assumed to be trusted, communication adds a constant time that can be measured offline once and then reused. This approach allows us to consider only the checksum loop in the detection threshold.

7.3 Memory Region Inclusion Probability

To evaluate how resilient our approach is regarding minor modifications in memory region containing the VF code (e.g., bit flips), we estimate the probability that a particular location is never included into the checksum result. We assume that memory accesses are distributed uniformly. Each block contains a single random memory access that loads an aligned 32-bit integer. For 2,500,000 iterations and a total checksum size of 524,288 integers, the probability that a memory location is never included in the checksum result is negligible:

$$(1 - 1/524288)^{2500000} = 0.0085$$

7.4 Runtime Overheads

Figure 6 shows a performance evaluation of SAGE using a multilayer perceptron (MLP) as an example. It consists of two Linear layers with weights of size `784×100` and `100×10`, with a Relu layer in between. As the workload increases, the overhead associated with the non-standard SAGE communication protocol becomes less noticeable.

The main sources of overhead are data transfers and kernel launches. Figure 7 shows that the copy operation requires additional time, which is a linear function of the input data size due to the additional data transfer between the host-accessible GPU memory and the GPU memory allocated by the SAGE kernel. SAGE adds less than 5% extra execution time to user

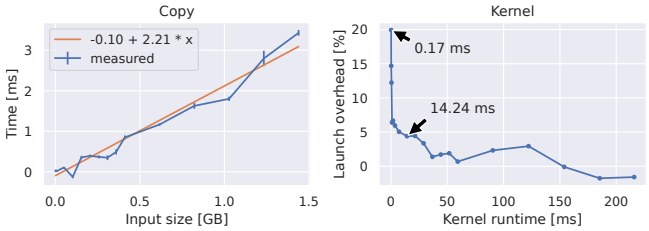


Figure 7: Overheads of data transfers and kernel launches.

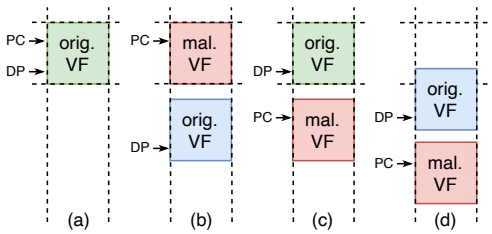


Figure 8: Memory copy attack variants.

kernels, which originally took more than 14.24 ms. Our evaluation does not take into account the overhead associated with the implementation of the encryption-decryption protocol, since its choice is left to the user.

7.5 Limitations of the Prototype

The use of self-modifying code requires triggering cache eviction of the instruction cache such that the modified instruction gets updated. With this cache eviction strategy, our implementation is able to achieve 75% of the maximum utilization. This is due to the GPU hardware not being able to load the required instructions in time for processing after the L2 cache eviction. If other cache eviction strategies become available to user code, higher utilization can be achieved. Unfortunately, triggering cache eviction using a large checksum loop limits the time difference caused by an adversary inserting instructions into the checksum loop. We believe that GPU vendors with in-depth knowledge of GPU architecture would be able to reduce the checksum loop size and use self-modification.

8 Security Analysis

In the following, we systematically analyze potential attacks given our threat model (see §3.3).

Pre-computation. The result of the checksum function depends on an unpredictable challenge issued by the verifier enclave. This prevents pre-computation attacks where the checksum value or part of the checksum (e.g., intermediate values) are pre-computed to later run code other than the VF.

Computation optimizations. The checksum function implementation must be time-optimal as algorithmic optimization would allow the adversary to find computationally faster or more efficient way of computing the checksum value (see §6.3 for details). SAGE is designed to prevent optimization, but to achieve provable guarantees the method of Gligor and Woo could be applied [8].

Attacks on the host system. The host system is untrusted (except for the verifier enclave) and the adversary is assumed to have administrative control over the system. This enables the adversary to eavesdrop, intercept, modify, or delay challenges or checksum results being transmitted between the verifier and the device. Given that the communication channel during the checksum computation is unauthenticated, the adversary could also inject challenges or checksum results. Modifications to the challenge would lead to a different checksum result. By injecting challenges the adversary could treat the VF as an oracle; however, given the unpredictable challenge generation, the probability of the verifier reusing the same challenge value is negligible.

Attacks on the device / Resource takeover. Before running the verification function, the device is considered untrusted. An adversary could be present on the device and interfere with the execution of the VF (e.g., by replacing or reordering instructions). This is prevented by the self-verification property and the strongly-ordered design of the checksum function. A strongly-ordered function requires the adversary to perform the same operations on the same data in the same sequence as the original function to obtain the correct result. Otherwise, the output differs with high probability if operations that have dependencies among them are evaluated in a different order.

Our design uses all available SMs simultaneously and maximizes thread and register usage. Thus, if an adversary would run a computation, the checksum computation would be deferred resulting in a considerable time overhead. However, the execution of a user kernel might not require all available GPU resources and would allow the adversary to take over these available resources. To prevent such attacks we require the user to develop kernels to achieve maximum GPU occupancy.

Memory copy attacks. Seshadri et al. [32,33] specify memory copy attacks that can be conducted by the adversary in the following three different ways as illustrated in Figure 8:

(b) The adversary replaces the checksum function with an altered checksum function and executes it, but computes the checksum over a correct copy of the checksum function elsewhere in memory. Thus, the program counter is correct, but the data pointer points to the original copy of the checksum function in a different memory location.

(c) The adversary uses the correct checksum function code in the original memory location to compute the checksum value, but executes a modified checksum function elsewhere in memory. Thus, the data pointer points to the original checksum function, but the program counter will be different.

(d) The adversary places both the original checksum function code and its altered version elsewhere from the memory locations where the correct checksum code originally resided. Thus, both the program counter and the data pointer will be different compared to an execution of the original checksum function.

To prevent memory copy attacks, both the program counter and the data pointer need to be included in the computation of the checksum. The DP is included in each step of the

computation, whereas the PC is indirectly included using self-modifying code. In addition to these specified attacks, the attacker could also copy the entire checksum function to a different location. This will not lead to a successful attack because the absolute location of the checksum function is irrelevant to security as long as the function pointer and the data pointer remain the same relative to the location of the effective checksum function in memory.

Proxy attacks. We refer to proxy attacks as attacks where the adversary eavesdrops on the communication and obtains the challenge sent to the device, sends it to a proxy, computes the checksum function there and returns the result to the verifier. We distinguish between the following cases. *GPUs on the same host*: we establish (and maintain) root-of-trust in sequence starting from the most powerful GPU to the least powerful one. *GPUs on a different host*: by involving a remote entity, the measured execution time will increase by the network latency for both sending the challenge and receiving the response. Tuning the number of checksum iterations to make the detection threshold smaller than the network latency, prevents using a more powerful GPU in a remote location.

Time-of-check to time-of-use attacks [5] / Execution environment takeover. In SAGE, these attacks are considered because the checksum computation happens prior to the execution of the user kernel. In particular, the adversary has two points where it could take over the execution environment set up by the VF: 1) before the launch of the user kernel, and 2) after the execution of the user kernel has completed. The former case is prevented by launching the user kernel(s) from within the VF epilogue. In the latter case, the execution of the user computation has finished and thus the user is indifferent whether the dynamic root-of-trust has been compromised. If the user wants to execute another kernel, the dynamic RoT needs to be re-established.

Replay attacks. To protect against duplicate transmissions of encrypted code and data between the SGX enclave and GPU, we add sequence numbers to each transmission.

Execution integrity and memory protection. While code and data are encrypted in transit, they have to be decrypted before use and placed into GPU memory. We control the allocation of memory from the kernel caller by calling `malloc()` from the device code. CUDA guarantees that the memory allocated in this way (unlike `cudaMalloc()`) is not accessible from the CUDA runtime or driver API. Therefore, even an attacker with root access to the operating system will not be able to access application code and data from such areas.

8.1 Formal Verification of Modified SAKE

To show that our modified SAKE protocol securely establishes a key between the verifier and the GPU, we have formally modeled the key establishment protocol and verified its security properties using the Tamarin prover [38] under the assumption that the computed checksum provides a short-lived secret. To model this property in Tamarin, we use a single-use authentic channel over which we send $w_2, \mathrm{MAC}_c(w_2)$. We show that the established symmetric key remains secret and is unique, a weak agreement exists between the verifier and the device, and recent liveness for each run of the protocol [37].

9 Related Work

To support trusted execution on GPUs, the following approaches were proposed. Graviton [46] specifies an architecture for supporting trusted execution environments on GPUs by changing the GPU's command processor to perform remote attestation based on device specific keys and ensure isolation between multiple processes running on the GPU. This is achieved by utilizing a set of keys where the root key gets embedded into the hardware of the device upon its creation. The latter requires modification to the GPU hardware by modifying the GPU's internal command processor to impose a strict ownership discipline.

HIX [13] proposes a heterogeneous isolated execution environment. HIX does not require modifications to the GPU architecture to offer an isolated execution environment, but instead physically modifies the I/O interconnect between the CPU and GPU and refactors the GPU device driver to work from within a TEE on the host. The TEE can then allocate trusted enclaves on the GPU.

HETEE [49] is based on a standalone computing system to dynamically allocate accelerators (such as GPUs or FPGAs) for either secure computing, or available to the host OS using PCIe switches. The security controller (and its software) is assumed to be trusted and interacts with the management CPU to control PCIe switching. HETEE attempts to provide isolation by selectively making accelerators available to specific applications by controlling communication to the accelerator through the security controller.

Telekine [9] illustrates side-channel attacks against TEE on GPUs based on observing the timing of GPU kernel execution. It then introduces a GPU stream abstraction that ensures execution and interaction through untrusted components are independent of any secret data. Telekine requires a GPU TEE to be deployed.

Machine learning represents a major use case for using GPUs as accelerators and can require privacy-preserving approaches for sensitive data. Slalom [42] uses a combination of a trusted enclave and untrusted GPU. The system decomposes the machine learning into two parts, where the control flow part runs inside the trusted enclave and operations that are not privacy sensitive (such as convolutions based on matrix multiplications) are offloaded to the GPU. Unfortunately, the split results in a decrease of training and inference accuracy.

SOTER [35] relies on the associativity property of operators present in DNN models. It assumes that the GPU is untrusted and sends modified parameter data from the SGX enclave. The output data received from the GPU is converted again to produce the expected result. To check the integrity of

the result, SOTER creates challenges and expects the GPU to return a proof of computation. If the integrity is compromised, the proof will be incorrect.

9.1 Hardware-based Attestation

NVIDIA has introduced a *confidential computing* [26] feature in the Hopper architecture. While technical details are currently scarce, this feature is touted to require no changes to the application code, while ensuring both the confidentiality and integrity of data and code running on the device.

With the addition of a vendor-backed hardware TEE solution, there is a question regarding the relevance of software-based attestation. While a hardware implementation provides reasonable levels of security, there have been several examples where the hardware-based techniques were flawed [6, 28, 44]. In these cases software-based techniques could come to the rescue. Software attestation can be complementary to the newly-added confidential computing feature and add another level of security to achieve defense in depth. Further, since the software layer doesn't rely on the private keys embedded in hardware by the manufacturers, it also reduces the TCB.

The trust required to obtain the properties provided by attestation is further reduced by combining both hardware- and software-based approaches. In essence, as long as one of the attestation methods is secure, the properties obtained using attestation hold.

9.2 Software-based Attestation

SWATT [33] uses a verification function that is based on pseudo-random memory traversal to compute the checksum. The verifier measures the execution time and verifies the checksum. Malicious code is required to verify each memory access to replace memory reads of changed locations with expected content, resulting in detectable time overhead. SWATT checks the entire memory of a system and its running time becomes prohibitive on systems with large memories.

PIONEER [32] verifies the integrity and guarantees the execution of code using a checksum function that is closely tied to the Pentium 4 architecture. The checksum function computes a fingerprint of the verification function and sets up an untampered execution environment. It is constructed such that manipulations by the adversary will noticeably increase the computation time.

Kovah et al. [16] and Butterworth et al. [3] extended the checksum computation to work on a Microsoft Windows system (CPU only), enabling a remote verifier to attest to a running system in a corporate environment.

Shaneck et al. [34] describe a software-based approach to remotely attest the static memory contents of sensors without requiring any additional hardware on the sensors nor precise measurements of execution timing. They use self-modifying code that generates memory read and jump instructions during the execution of their code.

Gligor and Woo [8] proposed a system that allows to provably establish a root of trust and provide secure initial states for all software unconditionally. The authors design a family of k-independent (almost) universal hash functions based on polynomials and use Horner's rule to show time- and memory-optimal evaluation of polynomials. An interesting area of future work is to translate these results to the context of computation on GPUs.

10 Conclusion

The prospect of software-only trust root establishment and secure code execution on GPUs offers exciting opportunities: execution of sensitive GPU code that should not be leaked to the GPU operator (code secrecy), correct execution of GPU code in an adversarial environment (code and execution integrity), preserving data correctness and confidentiality in the presence of malicious code on the system (data secrecy and integrity). SAGE represents a first step for achieving these properties on the NVIDIA Ampere architecture, under the circumstances that the architectural details about the Ampere architecture are closed-source. Since architectural knowledge for designing the verification function (VF) is key, our software-based approach to provide secure code execution on GPU paves the way forward for GPU vendors: they are naturally in a position to align the design of the VF to their architectural knowledge and lead the standardization process for trust establishment on GPUs.

Remaining open challenges include the design of software-based secure execution on alternative platforms, improving the execution speed of the verification function, and extend the execution model to support libraries that use a hybrid CPU+GPU compute model (e.g., TensorFlow [41]). Ultimately, an interesting future research question to answer is the interplay between hardware- and software-based approaches for trusted execution to achieve the strongest possible security properties for GPU-based execution.

Acknowledgments

This work was supported by the European Union's HE research and innovation programme under the grant agreement No. 101070141 (Project GLACIATION). We thank CSCS for providing access to compute resources used for this work.

References

[1] Ross Anderson, Francesco Bergadano, Bruno Crispo, Jong-Hyeon Lee, Charalampos Manifavas, and Roger Needham. A new family of authentication protocols. *ACM SIGOPS Operating Systems Review*, 32(4):9–20, 1998.

[2] ARM. ARM TrustZone technology. https://developer.arm.com/ip-products/security-ip/trustzone, 2021. [Online; accessed 01-Feb-2022].

[3] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. BIOS chronomancy: Fixing the core root of trust for measurement. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, November 2013.

[4] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[5] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the TOCTOU problem in remote attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2921–2936, 2021.

[6] Mark Ermolov and Maxim Goryachy. How to hack a turned-off computer, or running unsigned code in Intel Management Engine. *Black Hat Europe*, 2017.

[7] Bruno Forlin, Ronaldo Husemann, Luigi Carro, Cezar Reinbrecht, Said Hamdioui, and Mottaqiallah Taouil. G-PUF: An intrinsic PUF based on GPU error signatures. In *IEEE European Test Symposium (ETS)*, pages 1–2, 2020.

[8] Virgil D Gligor and Shan Leung Maverick Woo. Establishing software root of trust unconditionally. In *NDSS*, 2019.

[9] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 817–833, 2020.

[10] Intel. Interface for generic crypto library APIs required in SDK implementation. https://github.com/intel/linux-sgx/blob/master/common/inc/sgx_tcrypto.h, 2021. [Online; accessed 01-Feb-2022].

[11] Intel. Software Guard Extensions for Linux. https://github.com/intel/linux-sgx, 2021. [Online; accessed 01-Feb-2022].

[12] Intel. Xeon Gold 6348 Processor. https://ark.intel.com/content/www/us/en/ark/products/212456/intel-xeon-gold-6348-processor-42m-cache-2-60-ghz.html, 2021. [Online; accessed 01-Feb-2022].

[13] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 455–468, 2019.

[14] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[15] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[16] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *IEEE Symposium on Security and Privacy (SP)*, May 2012.

[17] George Marsaglia. DIEHARD: A battery of tests of randomness. 1996.

[18] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding PCIe performance for end host networking. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[19] NVIDIA. Ampere architecture in-depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/, 2020. [Online; accessed 01-Feb-2022].

[20] NVIDIA. How NVIDIA EGX is forming central nervous system of global industries. https://blogs.nvidia.com/blog/2020/05/15/egx-security-resiliency/, 2020. [Online; accessed 01-Feb-2022].

[21] NVIDIA. CUDA binary utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html, 2021. [Online; accessed 01-Feb-2022].

[22] NVIDIA. CUDA occupancy calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html, 2021. [Online; accessed 01-Feb-2022].

[23] NVIDIA. INT 32 and FP64 can be used concurrently in the Volta architecture? https://forums.developer.nvidia.com/t/int-32-and-fp64-can-be-used-concurrently-in-the-volta-architecture/108729/4, 2021. [Online; accessed 01-Feb-2022].

[24] NVIDIA. Multi-instance GPU user guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/, 2021. [Online; accessed 01-Feb-2022].

[25] NVIDIA. Basic linear algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas, 2022. [Online; accessed 01-Feb-2022].

[26] NVIDIA. NVIDIA H100 Tensor Core GPU architecture. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper, 2022. [Online; accessed 12-Dec-2022].

[27] Lena E Olson, Jason Power, Mark D Hill, and David A Wood. Border control: Sandboxing accelerators. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 470–481. IEEE, 2015.

[28] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867. IEEE, 2021.

[29] Research and Markets. Data center accelerator market – global forecast to 2026. https://www.researchandmarkets.com/reports/5390148/data-center-accelerator-market-by-processor-type, 2021. [Online; accessed 01-Feb-2022].

[30] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Boris Škorić, Stefan Katzenbeisser, and Jakub Szefer. Decay-based DRAM PUFs in commodity devices. *IEEE Transactions on Dependable and Secure Computing*, 16(3):462–475, 2019.

[31] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software attestation for key establishment in sensor networks. In *International Conference on Distributed Computing in Sensor Systems*, pages 372–385, 2008.

[32] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the ACM symposium on Operating systems principles*, pages 1–16, 2005.

[33] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy (SP)*, pages 272–282, 2004.

[34] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In *European Workshop on Security in Ad-hoc and Sensor Networks*, pages 27–41. Springer, 2005.

[35] Tianxiang Shen, Ji Qi, Jianyu Jiang, Xian Wang, Siyuan Wen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Xiapu Luo, Fengwei Zhang, and Heming Cui. SOTER: Guarding black-box inference for general neural networks at the edge. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 723–738, Carlsbad, CA, July 2022. USENIX Association.

[36] Elaine Barker Smid, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Special Publication (NIST SP), National Institute of Standards and Technology. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=906762, 2010. [Online; accessed 01-Feb-2022].

[37] Tamarin Team. Tamarin manual - property specification. https://tamarin-prover.github.io/manual/book/007_property-specification.html, 2021. [Online; accessed 01-Feb-2022].

[38] Tamarin Team. Tamarin prover. https://tamarin-prover.github.io/, 2021. [Online; accessed 01-Feb-2022].

[39] TechSpot. Intel's SGX deprecation impacts DRM and Ultra HD Blu-ray support. https://www.techspot.com/news/93006-intel-sgx-deprecation-impacts-drm-ultra-hd-blu.html, 2022. [Online; accessed 01-Feb-2022].

[40] Je Sen Teh, Azman Samsudin, Mishal Al-Mazrooie, and Amir Akhavan. GPUs and chaos: A new true random number generator. *Nonlinear Dynamics*, 82(4):1913–1922, 2015.

[41] Tensorflow. An end-to-end open source machine learning platform. https://www.tensorflow.org/, 2021. [Online; accessed 01-Feb-2022].

[42] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[43] Pol Van Aubel, Daniel J Bernstein, and Ruben Niederhagen. Investigating SRAM PUFs in large CPUs and GPUs. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 228–247, 2015.

[44] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural Load Value Injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72, 2020.

[45] Sebastiano Vigna. Further scramblings of Marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*, 315:175–181, 2017.

[46] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 681–696, 2018.

[47] John Walker. ENT – A pseudorandom number sequence test program. https://www.fourmilab.ch/random/, 2008. [Online; accessed 01-Feb-2022].

[48] Jun Zhao, Virgil Gligor, Adrian Perrig, and James Newsome. ReDABLS: Revisiting device attestation with bounded leakage of secrets. In *Cambridge International Workshop on Security Protocols*, pages 94–114, 2013.

[49] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Lutan Zhao, Fengkai Yuan, Peinan Li, Zhongpu Wang, Boyan Zhao, et al. Enabling privacy-preserving, compute-and data-intensive computing using heterogeneous trusted execution environment. *arXiv preprint arXiv:1904.04782*, 2019.

Confidential Computing within an AI Accelerator

Kapil Vaswani[1], Stavros Volos[1], Cédric Fournet[1]
Antonio Nino Diaz[1], Ken Gordon[1], Balaji Vembu[3,†], Sam Webster[1], David Chisnall[1], Saurabh Kulkarni[4,†]
Graham Cunningham[5,‡], Richard Osborne[2], Daniel Wilkinson[6,‡]

[1]*Microsoft* [2]*Graphcore* [3]*Meta* [4]*Lucata Systems* [5]*XTX Markets* [6]*Imagination Technologies*

Abstract

We present IPU Trusted Extensions (ITX), a set of hardware extensions that enables trusted execution environments in Graphcore's AI accelerators. ITX enables the execution of AI workloads with strong confidentiality and integrity guarantees at low performance overheads. ITX isolates workloads from untrusted hosts, and ensures their data and models remain encrypted at all times except within the accelerator's chip. ITX includes a hardware root-of-trust that provides attestation capabilities and orchestrates trusted execution, and on-chip programmable cryptographic engines for authenticated encryption of code/data at PCIe bandwidth.

We also present software for ITX in the form of compiler and runtime extensions that support multi-party training without requiring a CPU-based TEE.

We included experimental support for ITX in Graphcore's GC200 IPU taped out at TSMC's 7nm node. Its evaluation on a development board using standard DNN training workloads suggests that ITX adds $< 5\%$ performance overhead and delivers up to 17x better performance compared to CPU-based confidential computing systems based on AMD SEV-SNP.

1 Introduction

Machine learning (ML) is transforming many tasks such as medical diagnostics, video analytics, and financial forecasting. Their progress is largely driven by the computational capabilities and large memory bandwidth of AI accelerators such as NVIDIA GPUs, Alibaba's NPU [2], Google's TPU [18], and Amazon's Inferentia [3]. Their security and privacy is a serious concern: due to the nature and volume of data required to train sophisticated models, the sharing of accelerators in public clouds to reduce cost, and the increasing frequency and severity of data breaches, there is a realization that machine learning systems require stronger end-to-end protection mechanisms for their sensitive models and data.

[†]Work done while at Microsoft; [‡]Work done while at Graphcore.

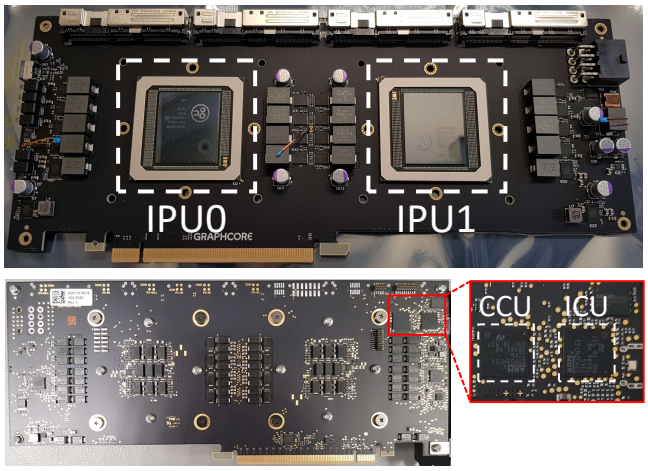


Figure 1: Graphcore Intelligence Processing Unit (IPU) development board (May 2020) with ITX extensions, showing two IPUs on the front side connected to the CCU via the ICU on the back.

Confidential computing [1, 4, 11, 31] relies on custom hardware support for trusted execution environments (TEE), also known as enclaves, that can provide such security guarantees. Abstractly, a TEE is capable of hosting code and data while protecting them from privileged attackers. The hardware can also measure this code and data to issue an *attestation report*, which can be verified by any remote party to establish trust in the TEE. In principle, confidential computing enables multiple organizations to collaborate and train models using sensitive data, and to serve these models with assurance that their data and models remain protected. However, the predominant TEEs such as Intel SGX [22], AMD SEV-SNP [5], Intel TDX [16], and ARM CCA [6] are limited to CPUs. Recently, NVIDIA has announced TEE support in upcoming Hopper GPUs [25] that works in conjunction with CPU TEEs.

Adding native support for confidential computing into AI accelerators can greatly increase their security, but also involves many challenges. Security features such as isolation, attestation, and side-channel resilience must be fitted in their

highly optimized architecture, with minimal design changes, and without degrading their functionality, performance, or usability. An additional requirement is the flexibility to operate with various hosts, including CPUs with no TEE support, CPUs with process-based TEEs such as Intel SGX, and CPUs with VM-based TEEs such as AMD SEV-SNP. None of the accelerator TEE designs that have been proposed meet this requirement, including NVIDIA GPU TEEs. Finally, the manufacturing and assembly process and protocols must be hardened against supply chain attacks.

This paper describes our effort to support TEEs in Graphcore's GC200 Intelligence Processing Unit (IPU), a state-of-the-art custom AI accelerator. We introduce *IPU Trusted Extensions* (ITX), a set of experimental hardware capabilities in IPUs. We show that, using ITX in conjunction with appropriate compiler and runtime support, we can delegate ML tasks to the IPU with strong confidentiality and integrity guarantees while delivering accelerator-grade performance. In particular, ITX can guarantee isolation of an ML application from an untrusted host: application code and data appears in cleartext only within the IPU, and remains encrypted otherwise, including when transferred over the PCIe link between the host and the IPU. Once an application is deployed within an ITX TEE, the host can no longer tamper with the application state or the IPU configuration. ITX can also issue remotely verifiable attestations, rooted in a Public Key Infrastructure (PKI), enabling a relying party to establish trust in a given ML task before releasing secrets such as data decryption keys.

The main components of ITX are a new execution mode in the IPU for isolating all security sensitive state from the host and securely handling security exceptions, programmable cryptographic engines capable of encrypting/decrypting CPU-IPU PCIe traffic at line rate (32 GB/s bidirectional throughput supporting PCIe Gen4), and a novel authenticated encryption protocol for ensuring confidentiality and integrity of code/data transfers without requiring trust in the host.

Trust in ITX is rooted in the *Confidential Compute Unit (CCU)*, a new hardware Root-of-Trust (HRoT) on the IPU board. The CCU provides each device with a unique identity based on a hardware secret sampled within the CCU at the end of manufacturing. The CCU firmware is responsible for managing the entire lifecycle of TEEs on the IPU, including creation, issuing attestation reports that capture IPU and task specific attributes, key exchange, launch, and termination of TEEs. Our design also features protocols for securely provisioning firmware to the IPU in a potentially hostile manufacturing environment, for issuing certificates that capture the identity of all updatable firmware, and for supporting firmware updates without requiring device re-certification.

Several distinguishing aspects of ITX and the IPU programming model result in stronger security than one may expect from CPU-based TEEs:

- An ITX TEE spans the entire IPU, and has exclusive access to all IPU resources until it terminates. Therefore, it is not possible for an adversary to run concurrently on the same resources and exploit the resulting side-channels. This execution model is feasible as most AI workloads require at least one accelerator, with larger workloads requiring thousands of accelerators for hours.

- The IPU memory system consists of large on-chip SRAM attached to its cores, which is loaded with data from untrusted external memory during explicit synchronization phases. Thus, during computational phases, code and data accesses to IPU memory have a fixed latency. This has two security implications: (1) traffic between IPU cores and memory need not be encrypted, as it stays within the chip; (2) this avoids the need for optimizations such as caching or speculation to hide memory access latency, and the resulting side-channels.

- The IPU supports a programming model where allocation and scheduling of all resources on the IPU (cores, memory, and communication channels) are statically managed by the compiler. Hence, the IPU application binary defines its entire data and control flow, including data transfers within the IPU, and between IPU and host memory. This is unlike GPUs where the host software stack (runtime and driver) remain in full control of the execution, and must be trusted to guarantee integrity.

There are many ways for software to utilize ITX to provide end-to-end guarantees for ML workloads, depending on the threat model and capabilities of the host. This paper focuses on configurations where a multi-party ML training workload is deployed to the IPU *without trusting the host CPU*. This mode has the strongest security properties and can be used with any CPU. We describe a prototype software stack and protocols for it, and present its end-to-end evaluation using standard DNN training workloads. Software to support other configurations, e.g., where the IPU is coupled with a hardware-protected CPU TEE, are left for future work.

We have fully implemented ITX in the IPU, taped out in 2020 and manufactured in TSMC's 7nm node. Our extensions use less than 1% of this large ASIC, and do not require any changes to its compute core or memory subsystem. Its evaluation on a development board using confidential ML training workloads suggests a performance overhead of less than 5% compared to non-confidential IPU workloads. While our prototype demonstrated promising results, significant work remains to turn our work into production.

Due to implementation constraints, our prototype uses a discrete HRoT (instead of an on-die core) and it does not encrypt traffic over IPU-IPU links. It is therefore vulnerable to physical attacks, e.g., on the link between the CCU and IPU, or between multiple IPUs. These vulnerabilities are not limitations of our design and can be addressed in future IPU generations by integrating the HRoT on the IPU chip, and by introducing encryption engines on IPU-IPU links.

In summary, this paper makes the following contributions:

1. A set of experimental hardware extensions to the IPU, a commodity custom AI accelerator, that enable high-performance confidential multi-party machine learning.
2. Support for remote attestation and secure key exchange based on a discrete hardware root-of-trust.
3. A pipelined application-level protocol for authenticated encryption & decryption of code and data over PCIe.
4. Protocols for securely provisioning secrets, firmware, and certificates to a device during manufacturing.
5. Prototype software support for enabling confidential multi-party training of ML models expressed in TensorFlow on the IPUs without requiring trust in the CPU.
6. Hardware implementation of ITX in the IPU ASIC manufactured by TSMC in 7nm node, and its initial evaluation on a development board in 2020, suggesting low overheads and orders of magnitude improvements over CPU TEEs. This made our prototype the first AI accelerator to support confidential computing.

While some aspects of our design are specific to IPUs, we hope it can serve as a blueprint for adding TEE support in other specialized devices and accelerators.

2 Background

This section outlines the IPU architecture and programming model, focusing on aspects relevant to security. The section also reviews hardware-based confidential computing.

2.1 IPU Hardware Architecture

Tiles. Each IPU consists of a set of *tiles*, each with a multi-threaded core and a small amount of private on-chip SRAM. The IPU features 1472 tiles, totalling roughly 900 MB of on-chip SRAM. The cores support an instruction set tuned for AI, including specialized vector instructions and low-precision arithmetic. Each core can execute up to six statically scheduled threads. Since on-chip memory is accessed at fixed latency, instructions can be exactly scheduled by the compiler.

Interconnects. The tiles are connected over *internal exchange*, an all-to-all, stateless, synchronous and non-blocking high-bandwidth interconnect whose operation is similarly orchestrated by software. The internal exchange is connected to an *external exchange* interconnect via a set of *exchange blocks*. Each exchange block manages a subset of the tiles and mediates traffic between the two interconnects. Each IPU has a pair of PCIe links that connect to a host server, and additional IPU-Links that connect to other IPUs.

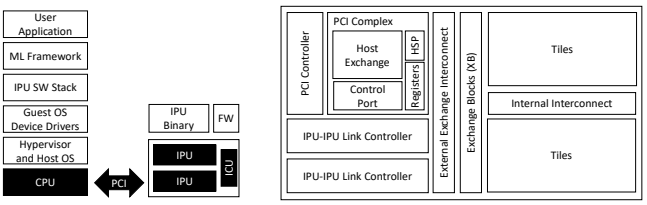


Figure 2: System stack (left) and IPU floorplan (right).

The external interconnect is a packet-switched Network-on-Chip. Tiles use the external interconnect to dispatch packets to the host via PCIe links and unicast/multi-cast packets to tiles on other IPUs via IPU-links. Tiles read data from the host by issuing a *read request* packet and waiting for all associated *read completion* packets. Tiles write data to the host by issuing one or more *write request* packets. Packets are routed based on tile identifiers. For requests, packets from exchange blocks are placed onto lanes based on the source tile identifier of the exchange packet. For read completions, the exchange lane is chosen based on the destination tile identifier, which is recorded in a lookup table in the PCI complex for each outstanding read request.

Exchange Address Spaces. The IPU exposes three address spaces to facilitate communication between the host and the IPU and between IPUs. The *Tile address space* is used by tiles to address one another. The *Host PCI space* is used by the host to address tile memory and on-chip page tables in the Host Exchange block. The *Tile PCI space* is used by tiles to address read requests to host memory over PCI. The IPU can be configured to re-map read requests from tiles to the PCI domain using on-chip page tables.

Host-IPU Interface. The IPU exposes a set of configuration registers to the host via a PCI BAR space. These registers are hosted in a component known as the *PCI Complex*. The PCI complex consists of a *Host Sync Proxy (HSP)* responsible for external synchronization between the host and the IPU, a *host exchange* that translates packets between PCI format and a proprietary external-exchange packet format, *on-chip page tables* for address translation of read/write requests from tiles to the PCI domain, on-chip lookup tables for keeping metadata for outstanding PCI read requests, and a *control port* that provides access to configuration registers of all other internal components.

The host exchange subsystem also includes a component known as the *autoloader*, which enables efficient scrubbing and initialization of tile memory. To initialize a binary in tile memory, the host can load small programs (e.g., a bootloader) into the autoloader, which can then broadcast it to all tiles.

Host-IPU Synchronization. The IPU execution model is based on the Bulk Synchronous Parallel (BSP) paradigm with barriers and supersteps. A superstep involves a global synchronization barrier between all tiles on one or more IPUs, followed by an exchange phase that transfers data between

tiles, followed by a compute phase which ends at another barrier. This process repeats until some application specific criteria is met—e.g., loss is under a threshold.

Using the HSP registers, the host can configure the frequency of synchronization barriers and indicate barriers at which it expects to be notified—e.g., when one or more batch of data has been processed, at epoch boundaries. Once configured, the IPUs can execute multiple supersteps independently without requiring involvement from the host.

IPU Control Unit (ICU). The ICU is a microcontroller integrated on the board and connected with the IPUs via JTAG, and with PCB peripherals for power supply and environmental monitoring. It is responsible for initialization of the IPUs.

Resets. The main means of resetting the IPU from the host is a *secondary bus reset (SBR)* that resets the entire device including the IPUs, the ICU, and the host link; the ICU must re-enable the host link once it comes out of reset. Alternatively, a *Newmanry Reset* can be triggered by writing the IPU control register; it resets the device logic including the host and IPU links, but does not reset the physical links. In both of these resets, tile memory is not scrubbed.

2.2 IPU Software Stack

The IPU software stack compiles and executes applications written in ML frameworks such as TensorFlow and PyTorch. It consists of a compiler, a host runtime, and a set of libraries supported by the IPU device driver.

Compiler. Given a computation graph representing a task (e.g., a TensorFlow XLA graph), the compiler partitions each layer of the graph between tiles, so that each tile holds a part of the model state (weights and activations for some layers) and a part of the input data. The compiler also assigns resources (threads, memory) to each node of the graph, schedules its computation, and finally emits specialized code for each tile.

The resulting IPU binary captures the different phases of execution, including I/O for reading batches of data, code for running the training loop, and I/O for writing the weights of the trained model. I/O phases also include synchronization and internal exchange code for exchanging data among tiles.

The compiler maps all data transfers between the host and IPUs to an abstraction called *streams* supported by the runtime. Data transfers from the host to an IPU (and IPU to the host) are mapped to input (output) streams and compiled to sequences of read (write) instructions to the Tile PCI address space. The compiler also uses streams to implement checkpoints: checkpoint creation maps all model weights to a single output stream, and checkpoint restoration reads them back from a single input stream.

The compiler supports an offline mode, which decouples compilation from execution. In this mode, the compiler generates self-contained IPU binaries, which can be persisted and loaded into one or more IPUs at a later point in time.

Host Runtime. The runtime provides code for loading IPU binaries, and for streaming data in and out of the IPUs. It loads IPU binaries by deploying a small bootloader into a reserved section of each tile memory. The bootloader in turn reads each tile-specific binary from the host into tile memory.

The runtime implements input streams by repeatedly copying data into a ring buffer in host memory and mapping the pages of the ring buffer into Tile PCI space in the on-chip page table. Once the ring buffer is ready and the mapping is defined, code on tiles can issue read requests. Similarly, output streams are implemented by copying data from the ring buffer to application memory.

2.3 Confidential Computing

Confidential computing is a paradigm where code/data remain protected from privileged attackers throughout their lifecycle: at rest, in transit, and *during use*. Central to confidential computing is the notion of a *trusted execution environment* (TEE) with two key capabilities: it can host an application in a hardware-isolated environment, which protects the application from any external access including access from privileged attackers; and it can issue remotely verifiable attestations that capture security claims about the application hosted in the TEE and the platform supporting the TEE. The attestations can be used by a relying party to gain trust in an application.

TEEs are supported by recent processors from Intel and AMD; ARM also recently defined a specification for supporting TEEs. There are broadly two classes of CPU TEEs: process-based and VM-based. Process-based TEEs (e.g., Intel SGX) are designed to isolate a user-space application from an untrusted operating system (both guest and host) and the hypervisor. VM-based TEEs (e.g., AMD SEV-SNP, Intel TDX) are designed to protect an entire guest VM from the host operating system and the hypervisor. TEEs offer varying degrees of protection from attackers with physical access to the CPU. Most TEE implementations assume that attackers can snoop on interconnects between the CPU package and external components (e.g., off-chip DRAM) and protect data by encrypting and integrity-protecting memory traffic.

Remote attestation is typically rooted in an on-die hardware root of trust (HRoT) with exclusive access to a unique device secret provisioned into one-time programmable fuses during manufacturing. During boot, the HRoT uses the secret to derive a device-specific identity key. This key typically endorses keys used for signing attestation reports. The corresponding public key is endorsed by the hardware manufacturer.

3 Threat Model

TEE hardware is subject to a variety of attacks throughout its lifecycle, from chip design and manufacturing up until the hardware is decommissioned.

Trust in TEEs is rooted in hardware, and consequently in the chip designers and their OEMs involved in designing and manufacturing the chips. Additional trust is also required in the infrastructure for issuing certificates to each chip, and for publishing the last known good version of firmware trusted computing base (TCB). While this is also the case with the IPU, we wish to minimize trust in the rest of the supply chain. Hence, we conservatively assume that attackers control the manufacturing and assembly process after tapeout, including the process of provisioning firmware and/or secrets to each device and harvesting their Certificate Signing Requests.

After deployment, we assume a strong adversary that controls the entire system software stack, including the hypervisor and the host operating system, and also has physical access to the host. The adversary can access or tamper with any code and data transferred between the host and the IPU, either in operating system buffers or over PCIe. The adversary can also tamper with device memory directly via the PCI BAR, or map the victim application's tile PCI address space to host-side memory controlled by the attacker. Information leakage through side-channels such as timing, power analysis, and physical probes on the IPU are generally out of scope. However, we wish to offer protection from side-channels based on memory access patterns, and from low-level integrity attacks such as glitching.

We trust the IPU and HRoT packages, and we assume that the adversary cannot extract secrets or corrupt state within the packages. In particular, the IPU includes trusted SRAM within the IPU tiles accessed only via on-chip channels.

With the current generation of IPUs, we make additional trust assumptions in the ICU, which provides connectivity between the hardware RoT and the IPUs, and in links between IPUs. We trust the ICU firmware and the physical links that connect the HRoT, the ICU and the IPU. These trust assumptions can be removed in subsequent generations of the IPU by placing the HRoT on the IPU die, and encrypting communication over IPU-IPU links.

The ML source script and configuration are trusted. The ML framework and the compiler are trusted for integrity of the computation—i.e., to compile the model defined in the ML script correctly into a manifest and IPU binaries. In multi-party configurations (involving parties that do not trust one another), these assumptions can be met by having all parties review the script and configuration for the workload, then confirm that they all locally compile to the same manifest and binaries. Each party is trusted with the integrity and confidentiality of the data streams they provide for the computation; in particular, honest parties are trusted to correctly encrypt their data streams with a fresh encryption key, and to release this key to IPUs only after verifying their attestation report.

In configurations that couple the IPU with a host CPU TEE (e.g., Intel SGX and TDX, AMD SEV-SNP), the CPU package is also trusted, along with any software hosted in the TEE.With process-based TEEs, the CPU-based software

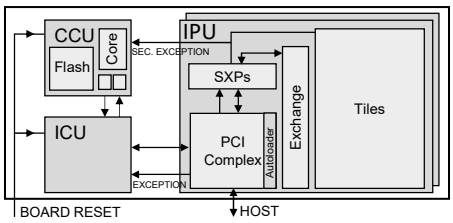


Figure 3: IPU hardware extensions to support trusted execution.

TCB may include the ML training or inferencing script and its framework (e.g., TensorFlow, PyTorch), compiler, and runtime. With VM-based TEEs the TCB may additionally include the kernel-mode driver and a guest operating system. The host runtime is trusted for confidentiality—i.e., to setup a secure, attested channel between the CPU TEE and IPU, and to transfer code/data over the channel.

Under this threat model, we wish to provide confidentiality and integrity guarantees for model code and data, including initial weights, input data, checkpoints and outputs. For training, integrity implies that the trained model is bitwise equivalent to the model obtained in the absence of the attacker. For inferencing, integrity implies that requests yield same results as those obtained in the absence of the attacker.

We wish to provide remote attestation, which refers to the ability of the platform to make remotely verifiable claims that a relying party can use to reason about the TEE's security properties and thereby establish trust in the application hosted within the TEE. Specifically, we wish to ensure that the attestation can deliver temporally fresh evidence that contains all security-sensitive parts of the platform and application state.

4 Overview

Trusted execution in IPUs enables model developers to securely offload an ML job (training or inference) while protecting both its model and data from the hosting platform. In turn, model developers can prove to data providers that their data remains protected from both the hosting platform and the model developers themselves. (Appendix A.1 provides a comprehensive security analysis of ITX.)

The workflow for offloading a job involves TEE creation, generation of an attestation report, its verification by remote parties, code/data encryption, secure exchange of encryption keys, job execution, and decryption of the outcome.

4.1 Hardware Extensions (ITX)

The IPU hardware contains several components (shown in Figure 3) to support this workflow, including a new hardware root-of-trust (RoT), called the CCU, and a new execution mode, called the *trusted mode*, in which all security sensitive state is isolated from a potentially malicious host. This mode

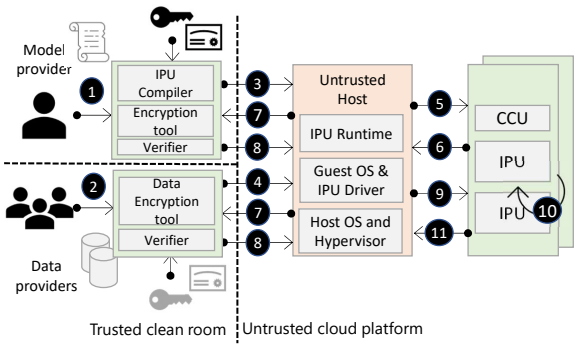


Figure 4: Multi-party training in trusted offline mode. Before training, the remote parties upload their encrypted code, data and certificates (1–4) Once training starts (5–6), they verify the attestation report (7) then release their encryption keys to the CCU (8-9); they can be offline for the rest of the computation. The IPUs train the model in a TEE (10) and releases an encrypted trained model (11), whose key can be shared with model receiver(s).

is entered by writing to a configuration register. (For remote verifiability, this register is measured by the CCU and included in the attestation report.) Once the IPU enters this mode, its configuration registers and tile memory can be accessed only by the CCU and ICU. The only way to exit this mode is via a chip reset, which is extended to scrub all key registers and tile memory.

The IPU also includes programmable AES-GCM engines for authenticated encryption and decryption of code and data transferred between the host and the IPU at PCIe line rate. These engines are hosted in new components, called *Secure Exchange Pipes (SXP)*, located on the interconnect between the PCIe block and the exchange blocks. The SXP and its use are described in Section 6.1.

4.2 Software Support

There are many ways to utilize ITX. For this paper, we illustrate a particular mode, called the *offline mode* (Figure 4). In this mode, a multiparty ML training workload can be deployed in an IPU-based TEE *without requiring a CPU-based TEE*. This mode has strong security properties (e.g., small TCB) and minimal dependencies on the host CPU hardware.

Job Preparation. In offline mode, a model developer uses an extended IPU compiler to statically compile a model training job expressed in an ML framework such as TensorFlow or PyTorch to standalone IPU binaries in a trusted, offline *clean room environment* (1). In addition to the binary, the compiler generates a *job manifest*, which contains auxiliary information required at runtime to execute the job. Next, the model developer encrypts binaries and parameters such as initial weights and learning rate using encryption keys that remain in the clean room environment. The model developer also generates a fresh public key share for key exchange, and a signature over the key share using their certificate. These artifacts, along with the model developer's certificate are packaged together to create an *application package*. Separately, data providers pre-process and encrypt their input data and labels in their own clean room environments, and create *data packages* which include their key shares and certificates (2). The resulting packages are uploaded to an IPU server (3, 4).

Job Initialization. Any entity can initiate execution of the training job using the host runtime, which has been extended to load encrypted code/data into IPUs. For confidential computing jobs, the runtime provides user-mode APIs for operations such as creating TEEs (5) for a job, fetching their attestation reports and additional collateral such as device-specific certificates (6), and relaying key-exchange messages from relying parties to the CCU (8). This runtime is not trusted.

Remote Attestation. In trusted mode, the CCU can issue remotely verifiable attestations, which are relayed to relying parties (7) as proof of TEE configuration for their workload. The attestation is a certificate chain from the IPU manufacturer root CA to an end-certificate signed by the CCU with custom extensions that embed initialization attributes (e.g., measurement of all security-sensitive IPU registers) and job-specific attributes, such as the measurement of the job manifest, and the hash digest of other runtime attributes, including certificate fingerprints of all parties and the CCU's fresh public keyshare. The model developer and data providers verify this report, the model, and identities of other participants. If they decide to make their data available for this job, they derive shared keys using the CCU's public key share and securely exchange their secrets with the CCU.

Job Execution. After the model developer and data providers have released their keys to the CCU, the CCU deploys the keys into the SXPs and starts the job (9) by installing a bootloader into the IPU tiles using the autoloader. The bootloader fetches the application binary from host memory to each tile in 1KB blocks. In trusted mode, these blocks are decrypted and integrity checked by the SXPs before being written to tile memory (see Section 7.3). Once the application binary has been transferred, the runtime initiates execution of the job. During execution, tiles generate read requests for data, also in blocks of 1KB. In trusted mode, the blocks are fetched from host memory over PCIe, and decrypted and integrity checked by the SXPs before being written to tile memory. Similarly, all write requests (e.g., checkpoints and trained model) are encrypted and extended with authentication tags before being written to host memory. The encryption protocol is mostly transparent to the compiler, which can compile *any training algorithm* into binary relying on the data being in tile memory in cleartext and utilizing all available IPU compute resources. Finally, the IPU encrypts the trained model with a key made available only to the model receivers listed in the job manifest.

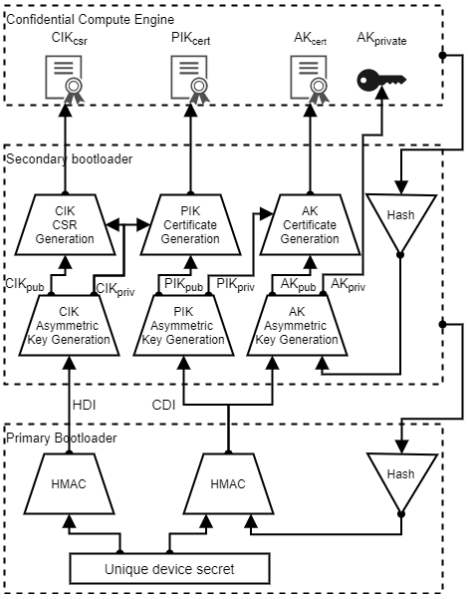


Figure 5: CCU firmware architecture and key hierarchy.

5 Trusted Execution on IPUs

5.1 Confidential Compute Unit (CCU)

The CCU is responsible for associating each device with a unique cryptographic identity and managing trusted execution in its IPUs. The CCU is a discrete chip based on STMicro's STM32H753 microcontroller [24]. This chip was selected as the RoT based on several security features required to implement measured boot and to offer protection from a variety of attacks throughout the IPU lifecycle, such as the ability to provision a custom bootloader during manufacturing and a mode that prevents external access via interfaces such as JTAG. As shown in Figure 3, the CCU is connected to the IPU via the ICU. A dedicated pin receives all exceptions generated by the IPU in trusted mode, giving the CCU firmware full control over exception handling. The CCU reset pin is coupled in hardware with the ICU reset pin and IPU reset, so they cannot be independently reset.

Firmware Architecture and Attestation. The CCU implements a measured boot protocol which is a variant of the Device Identity Composition Engine (DICE) architecture [12, 33]. DICE ensures that each device is assigned a unique identity while minimizing exposure of hardware secrets. Except for the stable device identity, all derived secrets and keys automatically change when firmware (and its measurement) changes, which ensures that low-level firmware attacks do not compromise secrets used within other firmware.

The CCU firmware (Figure 5) consists of three layers: an immutable primary bootloader provisioned in one-time programmable flash memory at manufacturing; a mutable secondary bootloader responsible for device identity and attestation certificates; and a confidential compute engine (CCE) that manages the TEE lifecycle.

During manufacturing, the CCU is provisioned with the primary bootloader firmware. When the device is brought out of reset for the first time, this primary bootloader receives control from ROM firmware, samples a *unique device secret* (UDS) using a hardware-based TRNG, stores it in a region of flash memory, and permanently blocks its access from any other firmware layers. The UDS is the root of the IPU key hierarchy, and this protocol ensures that it is never exposed outside the CCU, not even to the manufacturer.

On every subsequent boot, the CCU loads and authenticates the secondary bootloader from flash using the IPU manufacturer's firmware signing key. Next, it derives two intermediate secrets: a *Hardware Device Identifier* (HDI) from UDS, and a *Composite Device Identifier* (CDI) from UDS and the measurement of the secondary bootloader. HDI is unique to each card, while CDI is unique to each card and secondary bootloader. It then scrubs UDS from memory and transfers control to the secondary bootloader, handing over HDI and CDI.

The secondary bootloader further derives two public-private key pairs: a *Card Identity Key* (CIK) from HDI, and a *Platform Identity Key* (PIK) from CDI. Hence, CIK gives each card a stable identity whereas PIK is unique to each card and secondary bootloader. The bootloader also generates a self-signed CSR for CIK, a PIK CSR, and a PIK certificate signed by CIK. The PIK CSR and certificate contain a custom extension that records measurements of the secondary bootloader and the ICU firmware along with additional device-specific information. The CSRs can be securely harvested during manufacturing and endorsed by the IPU manufacturer CA, which issues CIK and PIK certificates. (See Appendix A.2.)

The secondary bootloader derives the *Attestation Key* (AK) from CDI and the CCE measurement. Hence, AK is unique to each device, secondary bootloader, and CCE. The bootloader issues an AK certificate with the CCE measurement in a custom extension, signed by PIK, and finally scrubs all secrets and transfers control to CCE, handing over AK.

The CCE uses AK to sign attestation reports containing IPU- and job-specific information (Section 5.2). A relying party can validate attestation reports using the device-issued AK certificate, and manufacturer-issued CIK/PIK certificates.

Firmware Update. Per DICE, a secondary bootloader update invalidates PIK certificates issued by the manufacturer and, as UDS is provisioned within each device, the IPU manufacturer cannot independently derive and certify the updated PIK. Instead, we rely on CIK, acting as a local CA, to sign the updated PIK certificate. Additionally, the manufacturer would issue TCB update certificates containing measurements of old and new versions of firmware. A relying party can validate attestations using device-issued PIK certificates, the original PIK and CIK certificates, and TCB update certificates. (See Appendix A.3–A.4 for more details and a hardened variant.)

5.2 TEE Lifecycle Management

The CCU exposes an API for TEE management on the IPU.

TEE Initialization. The first step in securely offloading a job to an IPU is to create a fresh TEE for this job. TEE initialization requires a job manifest (Appendix A.5), public key shares, signatures over the key shares and certificates for each relying party, and a checkpoint counter indicating whether the job is starting or resuming from a checkpoint. During TEE initialization, the CCU first *quiesces* the IPU, ensuring that there are no in-flight read and write requests between the host and IPU. It then switches the IPU into trusted mode, scrubs all tile memory using the autoloader, and measures the state of the configuration registers. It then checks the signatures over the key shares using the certificates, and generates its own fresh EC share, which is used to establish an ECDH shared secret between each relying party and the CCU.

The CCU generates an attestation report signed by the attestation key containing various IPU-specific attributes (e.g., configuration register measurements) and job-specific attributes such as the job manifest, certificate fingerprints for all parties, and the checkpoint counter consisting of the epoch counter and checkpoint identifier. (See Appendix A.6 for the details.)

Each relying party can review the attestation together with the supporting certificate chain, to validate the device and the initial state of the CCU and IPU, then it can compute its ECDH shared secret and wrap a key package that contains the party's data encryption keys and nonces to run the job. (See Table 2 in Appendix A.6 for the keying details.)

TEE Launch. After gathering wrapped encryption keys from all relying parties, the host launches the execution of a job.

First, the CCU computes the ECDH shared secret for each party and uses them to unwrap the key package(s) received from each party. It then combines the nonces to derive a checkpoint key and a final-model encryption key for this run of the job (and, if resuming from another run, the checkpoint key from that previous run to restore its state). This key derivation ensures both that the checkpoint key for this run is fresh (as long as one relying party's nonce is fresh) and that the checkpoint key of a prior run can be recomputed once all relying parties agree to resume from a checkpoint. (See Table 2 in Appendix A.6 for the keying details.)

Next, the CCU deploys a pre-defined bootloader on the IPU tiles using the autoloader, and it deploys a first set of encryption keys to the SXP (including the model key) as specified in the job manifest. It then activates the bootloader (whose measurement is included in the attestation report) on every tile, which issues requests to read their encrypted application binary from host memory. Responses to these read requests are authenticated and decrypted by the SXPs before being copied into private tile memory.

Finally, the CCU deploys the next set of encryption keys (including data keys) as specified in the job manifest, and triggers the main execution loop on the IPU tiles.

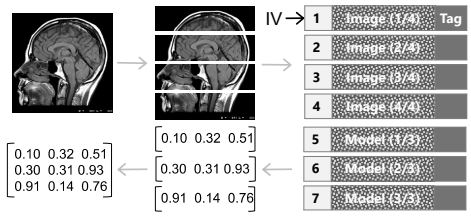


Figure 6: Authenticated encryption with explicit IVs. Data is partitioned into frames with unique IVs. Hardware-level decryption ensures their integrity based on their authentication tag; the receiver must verify that frame IVs match the expected IVs.

TEE Termination. At any point after initialization of a TEE, the host runtime can also request that the TEE be terminated. The CCU may also terminate the TEE in the event of a security exception raised from the IPU such as failure to authenticate a response of a read request. During TEE termination, the CCU quiesces the IPUs, scrubs tile memory using the autoloader, and disables all SXP keys. Finally, the CCU switches the IPU into normal mode via *Newmanry* reset.

A TEE may also be terminated by a hard reset of the device. In this case, all CCU state is cleared and the IPU reverts to normal mode. When it comes out of reset, prior to re-enabling the host links, the ICU scrubs tile memory to ensure that any secrets left over from a previous execution are erased before the host re-gains access to the device.

6 Encrypted Direct Memory Access

Next, we describe the ITX protocol for encrypted code and data transfers to/from IPU tiles. The protocol is application-level as opposed to transport-level. While it is transparent to ML frameworks, it relies on application software (e.g., the IPU compiler) to assign IVs for authenticated encryption and to program the tiles to securely load code, initial weights, training data, and save/reload checkpoints and results. The protocol is supported in IPU hardware by fully pipelined AES256-GCM engines for authenticated encryption at PCIe line rate. This choice results in simpler hardware, allows the IPU to be coupled with untrusted CPUs (or CPUs with varying TEE support) and retains the compiler's ability to maximize PCIe utilization by parallelizing data transfers across tiles.

6.1 Data Format

In the encryption protocol (illustrated in Figure 6), application software partitions each code and data stream into equally-sized encrypted *frames*. Each frame consists of a 128-bit IV, followed by a series of cipher blocks that carry the encrypted contents of the frame, and by a 128-bit authentication tag. Application software is free to use different frame sizes for different streams, as long as the total frame size (including IV and authentication tag) is a multiple of 128 bytes with a maximum of 1KB, which is the largest supported PCIe read. Application

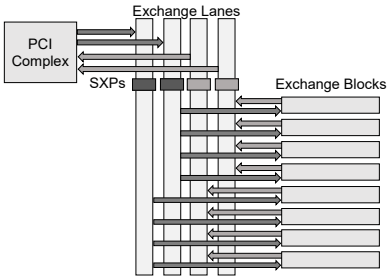


Figure 7: IPU External Exchange Interconnect, with an SXP on each exchange lane. Traffic is forwarded from and to exchange blocks to exchange lanes based on the exchange block identifier.

software can use different keys to encrypt different streams. This is critical for multi-party scenarios where streams are provided and accessed by different parties. Crucially, application software must ensure that IVs are never reused across frames encrypted with the same key, which would be catastrophic with AES-GCM. In our implementation, this invariant is ensured by the compiler, which constructs the IV by combining stream-specific identifiers and frame indexes, and the fact that both code and data streams are write-once abstractions. Together, this guarantees that unless the associated key has been compromised, authenticated decryption with the correct IV yields the correct payload.

6.2 Hardware Support

Multiple components in the IPU support ITX encryption. The IPU includes blocks, called *Secure Exchange Pipes*, extensions to packet formats for carrying encryption-related information, and extensions to exchange blocks and the PCIe complex for supporting the task of mapping frames to keys.

Secure Exchange Pipe (SXP). The SXP is a programmable hardware block that supports AES256-GCM authenticated encryption and decryption of frames. Each SXP achieves 16 GBps unidirectional throughput with negligible impact on latency. As shown in Figure 7, there are four SXPs placed on the exchange interconnect (two per direction) to support encryption/decryption at PCIe Gen4 line rate (32GBps bidirectional). In trusted mode, each SXP is configured to intercept read/write requests from four exchange blocks.

AES-GCM Engine. The SXP's core is a fully pipelined AES-GCM engine that supports 16 *physical key contexts* to enable concurrent requests. Each context can be programmed by loading a 256-bit key into control registers exposed to the CCU via an internal control bus. While frames may be interleaved, for functional correctness we require that *each context processes a single frame at a time*. This invariant is enforced by the compiler, as detailed in Section 7.1.

The core implements the standardized AES256-GCM algorithm with two restrictions: the additional authenticated data is always empty; and the plaintext is block-aligned and not empty. For convenience, we also treat the IV as a full 16-byte block, including the 32 bits of internal block counter. In each cycle, the core performs one of the following operations on its context: (i) the context is idle and the core receives the IV for the frame, (ii) the context is active and the core receives a block of data, or (iii) the context is active and the core receives a MAC. The core detects context switches by comparing key context identifiers between consecutive cycles, so that it can fetch the next context before the next operation.

Frame encryption/decryption. The SXP receives three types of external exchange packets: read requests (egress); read completions requiring decryption (ingress); and write requests requiring encryption (egress). Their headers are extended to carry additional information to help the SXP determine how the packets should be handled: an `AES` bit indicates that the read completion or the write request is encrypted; a 4-bit `KEY_INDEX` field identifies the physical key context to use; and a `CC` bit indicates the last packet of the frame and triggers the computation of its authentication tag.

In write request packets (outbound to the host PCIe domain), the `AES` and `CC` bits are set by the tile, whereas the `KEY_INDEX` is set by the SXP. In read completions packets, the information is set by the PCIe complex based on trusted state it maintains about pending read requests.

Read request packets and packets with the `AES` bit unset do not require encryption/decryption; they are passed unchanged. For all other packets, the header bypasses the AES core, then the AES core handles each packet (and its blocks) depending on whether a frame starts, a frame continues, or a frame ends.

Key Selection. Each SXP supports multiple physical key contexts to enable encryption/decryption of concurrent I/Os. The SXP provides a set of programmable (by CCU) registers to define a mapping between packets and the physical key context to use for encrypting/decrypting their payload. The compiler produces this mapping by assigning a set of tiles associated with an exchange block context to access a single stream. Upon receiving a packet, the SXP looks up the physical key context using the exchange block context index computed from the source tile identifier in the header.

Once the SXP infers the physical key context, it updates the `KEY_INDEX` field in the request packet header. For write requests, the field is then used by the SXP to switch the AES core to the inferred physical context for encrypting its payload. However, for read requests, the situation is more involved, as the read requests bypass the AES core, and the inferred physical key context must be used to decrypt the read completions that will be returned by the host after the read request has been processed. When the PCI complex receives the read request, it caches the `KEY_INDEX` and `AES` fields in an on-chip lookup table along with other metadata, such as the source tile identifier. When the corresponding read completions arrive from the host, the PCI complex retrieves these fields from this lookup tables and inserts them into the read completion packets. The SXP can then use these values to identify the physical key context to use for decrypting the payload. The

PCI complex tracks the number of pending read completion packets for each request, and sets the `CC` bit on the last one.

7 Software Extensions

We now describe a set of extensions to the IPU software stack to compile and execute confidential ML tasks using ITX in offline mode. This mode is triggered by a Tensorflow configuration option. When enabled: (1) The XLA backend transforms the computation graph to use a new abstraction called *confidential data streams* for all data transfers including initial weights, training data, checkpoints and the trained model. (2) The IPU compiler compiles the computation graph into a set of IPU application binaries (one for each IPU), where each binary is a concatenation of tile-specific binaries. The compiler encrypts tile binaries into a set of encrypted frames using a freshly sampled model key. A frame is assigned a unique IV comprised of code type, IPU/tile IDs and frame index. (3) The IPU runtime is extended to securely bootstrap the task, then transfer the encrypted binaries and data between the host and IPU. (See Appendix A.8 for a sample scenario.)

7.1 Confidential Data Streams

Confidential data streams is a compiler and runtime abstraction for transferring data to/from the IPU with confidentiality and integrity guarantees, leveraging SXPs. Each stream is a sequence of data instances encrypted with the same symmetric encryption key. Each data instance is partitioned into a sequence of frames, and each frame is encrypted using a unique IV composed of a stream type (data), a stream identifier, and the index of the frame within the stream.

The compiler and the runtime implement reads and writes to confidential data streams as follows. As discussed in Section 6.1, the compiler first assigns a region in tile PCI space to each stream, subject to the constraint that it never exceeds the total capacity of the IPU ring buffer (e.g., 256 MB).

Next, the compiler assigns sets of tiles to read from or write to each stream, reserves SRAM on each tile to hold a part of the stream, and generates SXP mappings, subject to the constraints that (a) the exchange block context associated with these tiles map to physical key contexts assigned to the stream, and (b) the number of physical key contexts in use at any point in the program does not exceed 16 for any SXP.

To maximize performance under these constraints, the compiler may introduce synchronization points in the application where existing keys are invalidated and new keys are loaded. The compiler includes these synchronization points in the job manifest, along with their key identifiers; and the (untrusted) IPU runtime uses this part of the manifest to ask the CCU to load the next decryption keys into the SXPs at these points. The key changes apply only to input streams. Keys for output streams are derived and loaded by the CCU at TEE launch, and do not change throughput its lifetime. A malicious runtime not following the job manifest's key schedule can only cause decryption failures, resulting at most in denial-of-service.

Next, the compiler schedules read/write operations on each tile. The schedule is required to satisfy a hardware constraint that, at any point, the tiles that generate requests targeting any given physical key context be associated with a single exchange block context. This is because, while the exchange block can dynamically synchronize and regulate requests within each exchange block context (so that its physical key context is used by one tile at a time) there is no such synchronization across exchange block contexts.

Finally, the compiler generates code on each tile that implements the schedule, to issue read/write requests for the accessed frames. (Details are omitted due to space constraints.)

7.2 Secure Checkpointing

Checkpoints are saved to and restored from host memory, and are implemented via data streams. Secure checkpointing is implemented via a special form of confidential data streams, where the IV captures the epoch (counting the number of checkpoint resumptions for this job) and the checkpoint identifier (counting the number of checkpoints stored within an epoch.) The CCU uses a separate checkpoint key for each epoch, and makes the epoch counter and checkpoint identifier available to IPU tiles via the bootloader at the start of the application. (See Appendix A.7 for more details.)

7.3 Secure Bootstrapping

Secure bootstrapping is the process of securely loading encrypted application binaries into the IPU, either at the start of a job, or while resuming a job from a checkpoint.

Bootstrapping involves the following steps. First, the IPU runtime loads the encrypted IPU binary in host memory and creates a TEE using the CCU APIs; this switches the IPU into trusted mode. Next, the CCU installs a *bootloader* (shown in Appendix A.5) onto every IPU tile using the autoloader described in Section 2.1, and also configures the SXPs with the model-decryption key. The bootloader on each tile fetches the tile's binary from host memory by issuing a sequence of read requests. Each frame received from the host is intercepted by the SXPs, authenticated and decrypted, and copied into tile memory. The bootloader then checks that the received IV matches the expected IV built into the bootloader logic; this check is performed in software because the SXPs only guarantee authenticity of each frame, not the integrity of the entire stream. Failure of this check indicates an attempt by the host to tamper with the code stream, such as by replaying/reordering frames. In such event, the tile raises a security exception, which is handled by the CCU. If all checks pass, the bootloader reconstructs the original binary by stripping IVs and authentication tags from all frames.

Finally, the bootloader computes a hash of the tile binary; the tiles accumulate a hash of the whole application; and the CCU checks that it matches the measurement in the job manifest, or generates a security exception otherwise. This protocol, together with bootloader integrity (its measurement is included in the attestation) guarantees application integrity.

8 Evaluation

Our evaluation focuses on TEE overheads for ML training when using CPUs and IPUs.

Implementation. We have implemented ITX on the IPU on a non-production development board. The IPU chip has been fabricated in TSMC's 7nm node, including the on-chip security extensions, which account for $< 1\%$ of the chip size. As part of post-fabrication validation, these extensions have been tested to verify they conform to their specified behavior.

We have integrated the CCU on the board and implemented the architecture described in Section 5, including the protocols for measured boot and TEE management.

We have implemented a software prototype for confidential training tasks where the host CPU server is untrusted. Our prototype includes experimental support in the ML framework, IPU compiler and runtime. There are a few gaps in our prototype: (1) our implementation currently supports only one IPU on the board; (2) the compiler makes use of only one logical key region onto which code, data, label, checkpoints, and outputs are mapped; nevertheless, every encrypted frame is statically assigned a unique IV, preserving the invariant that each IV is used only once; (3) secure resumption is not yet implemented; and (4) the bootloader deployed on IPU tiles does not measure the IPU binary after decryption.

Experimental Results. Figure 8 summarizes the hardware and software configuration of our testbeds. We evaluate the performance of confidential training on ResNet models of various sizes (20, 56, and 110) on the Cifar-10 dataset. The dataset consists of 60,000 32x32 images spanning 10 classes; 50,000 of these images are used for training the dataset and the remaining are used for testing the resulting model. We ran the same training code and data configurations in clear and confidential modes, and confirmed that they both yield models with the same prediction accuracy.

We compare IPU TEEs against CPU TEEs based on the largest available AMD SEV-SNP server. The early IPU development boards operate at reduced frequency of 900 MHz. The AMD CPU testbed utilizes 48 single-threaded cores; hyper-threading does not improve performance due to high vector unit utilization leaving little room for another hyper-thread. Scaling from 32 to 48 cores improved performance by 10%.

Figure 9 shows the training throughput that we achieve in clear and confidential modes. IPU-based training even with a single IPU operating at reduced frequency is 12-20x and 13-17x faster than CPU-based training in clear and confidential

Testbed	Training configuration
AMD SEV-SNP 48-core VM on EPYC 7763	ResNet-20. Batch size: 1534; 32 epochs. ResNet-56. Batch size: 768; 32 epochs. ResNet-110. Batch size: 384; 64 epochs.
ITX IPU @ 900 MHz, Intel Xeon 8168	ResNet-20. Batch size: 64; 32 epochs. ResNet-56. Batch size: 32; 32 epochs. ResNet-110. Batch size: 16; 64 epochs.

Figure 8: Testbed configuration for TensorFlow training of ResNet models on Cifar-10 dataset. In each configuration, batch sizes are optimized to yield maximum performance. (Smaller batches do not affect correctness, but may improve convergence or accuracy.)

Figure 9: Training throughput of ResNet models on cifar-10.

modes respectively. Enabling SEV-SNP introduces modest overheads, ranging from 8% (small model) to 14% (large model) while the overheads of enabling ITX range from 3% (large model) to 58% (small model). The ITX overhead is dominated by one-time setup cost, which is amortized over large training times; this cost includes TEE initialization and attestation (40%), TEE launch including SXP setup (43%) and TEE termination including SXP scrubbing (13%). Runtime encryption introduces only 4% of the total overhead. More generally, we expect the one-time cost to be negligible with state-of-the-art models, which take weeks or days to train. With ResNet-110 model, the overall overhead is just 3% (1123 vs 1089 seconds for running 64 epochs). We also expect that utilizing both IPUs at full frequency would deliver an additional performance improvement (up to 3.5x) over CPUs.

In summary, the evaluation shows that using ITX, AI workloads can continue to benefit from the use of accelerators without compromising on performance or security.

9 Discussion

Trusted CPU TEEs. While this paper mainly focuses on a configuration where IPUs are attached to an untrusted CPU, ITX supports configurations with varying trust in CPU TEEs.

For instance, ITX can be used in a configuration where IPUs are coupled with a process-based CPU TEE (e.g., Intel SGX) hosting TensorFlow, the IPU compiler and the IPU runtime, with the IPU kernel-mode driver and the OS running outside the TEE. In this configuration, the enclave would receive an encrypted model script from the model developer,

and the IPU runtime would encrypt the compiled IPU binary with fresh keys. Similarly, the enclave would receive encrypted training data from data providers on the basis of an Intel SGX and IPU attestation. Within the process-based CPU TEE, the data can be decrypted, pre-processed and aggregated (in parallel with job execution), and re-encrypted by the IPU runtime with fresh keys. Encrypted code/data still then need to be copied to a run buffer allocated outside the enclave (in the host process) accessible to the IPU. While this configuration has a larger TCB and incurs higher CPU cost, it offers greater flexibility and support for inferencing scenarios.

Inferencing. While this paper mainly focuses on training, ITX does support inferencing workloads. Scenarios, where the number of remote inference clients using a model is small and mostly static, can be served by the existing architecture (no trust in the CPU) without additional overhead by assigning a key context to each client. Scenarios, where the number of clients is large and dynamic, could be supported by introducing a trusted front-end server component (in a process-based TEE on the host CPU or a remote CPU) that terminates TLS connections, receives inference requests from the clients, and re-encrypts them in batches using a smaller number of key contexts on the IPU (to avoid frequent expensive re-keying in the IPU). This architecture still allows decoupling the choice of CPU TEEs and accelerator TEEs. This is unlike GPU TEEs that require the CPU and GPU TEEs to be on the same platform. Finally, the TCB of the re-encrypting TEE is relatively small and independent of the application.

Side-channels. Our design intends to minimize leakage from side-channels. On the IPU itself, leakage due to memory access patterns and timing is minimized for two reasons. First, computation on the IPU is statically scheduled by a trusted compiler. It is therefore possible to analyze a workloads at compile time to ensure that memory access patterns are data independent, or add padding otherwise. Second, the IPU cores do not rely on speculative execution and on-chip memory accesses incur a fixed latency. As a result, all I/O between the untrusted host CPU and IPUs occurs at fixed time intervals. Thus, the attacker can observe the time taken to process an entire batch, as opposed to time taken to process each layer in the model [15].

10 Related Work

Trusted hardware. There is a history of work [7, 8, 10, 13, 20, 20, 21, 27, 32, 35] on trusted hardware that isolates code/ data from the rest of the system. Intel SGX [22] and AMD SEV-SNP [5] are the latest in this line of work. Our work extends this approach from general-purpose CPUs to accelerators.

Trusted execution on accelerators. Our work is the first to demonstrate an ASIC with confidential computing capabilities and the only one that does not require trust in CPU TEEs.

NVIDIA recently announced confidential computing support in upcoming Hopper GPUs [25]. Their design shares the same core principles as ITX on IPUs. Hopper GPUs are equipped with an on-package hardware RoT responsible for attestation and enforcing course-grained GPU isolation under the assumption that on-package GPU memory is trusted. Hopper GPUs also support encrypted and integrity-protected communications (kernels and data) to and from the GPU. However, the NVIDIA design requires a VM-based CPU TEE as the responsibility of attesting and establishing a secure channel with the GPU lies within the kernel-mode driver.

Numerous mechanisms have been proposed to enable CPU TEEs to securely interact with I/O devices—e.g., GPUs [17, 34, 37], FPGAs [19, 26, 30, 38], and AI accelerators [14, 36, 39]. Some of this work has attempted to reduce trust on privileged host via hardware support on the GPU [34] or on the CPU [17]. Graviton [34] extends the GPU with support for secure resource management, and relies on a trusted GPU runtime hosted in a process-based CPU TEE to manage the TEE lifecycle. HIX [17] requires extensions to process-based CPU TEEs, including the PCI interconnect and the CPU's MMU. GuardNN has attempted to remove the CPU from the TCB [14] by introducing instructions for establishing a secure channel between remote users and the device, and for decrypting/encrypting inputs/outputs. However, such architecture does not guarantee integrity as the instruction schedule can be tampered by attackers controlling the CPU.

TEE-I/O. In parallel with our work, there has been an industry-wide effort to develop TEE-I/O, a standard framework for assignment of devices to VM-based CPU TEEs. This effort also includes the development of TEE Device Interface Security Protocol (TDISP [29]), an architecture for devices that support TEE-I/O. TDISP provides specifications for establishing trust between the VM-based TEE and the device (SPDM [9]), and for secure TEE-device communication (IDE [28]) and secure management of the device's lifecycle.

CPUs and devices that support TDISP are expected to be deployed in the next couple of years. Compared to application-level protocols (such as Section 6), TDISP is more efficient and transparent. CPUs that support TDISP provide hardware encryption for PCIe communication, removing the need for software encryption using explicit IVs. However, TDISP currently supports only VM-based TEEs, which brings the OS, device drivers, and other user-mode components in the TCB.

11 Conclusion

We presented ITX, a set of experimental hardware extensions for Graphcore IPUs. Our design provides application-level confidentiality and integrity for ML tasks offloaded to an untrusted cloud provider. We also presented a software architecture that removes trust from host CPUs, thereby minimizing the trusted computing base and removing dependencies on CPU TEEs. We implemented them in the GC200 IPU taped out at TSMC's 7nm node, and experimentally confirmed small performance overheads for training large models.

References

[1] Confidential computing consortium. https://confidentialcomputing.io/, 2022.

[2] Alibaba. Alibaba unveils AI chip to enhance cloud computing power. https://www.alibabacloud.com/blog/alibaba-unveils-ai-chip-to-enhance-cloud-computing-power_595409, 2022.

[3] Amazon. AWS Inferentia: High performance machine learning inference chip, custom designed by AWS. https://aws.amazon.com/machine-learning/inferentia, 2021.

[4] Amazon. Confidential Computing. https://aws.amazon.com/blogs/compute/tag/confidential-computing/, 2022.

[5] AMD. 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, 2021.

[6] ARM. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2023.

[7] Rick Boivie. SecureBlue++: CPU support for secure execution. 2011.

[8] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[9] DMTF. Security Protocol & Data Model (SPDM). https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.3.0.pdf, 2022.

[10] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry V. Ponomarev, Nael B. Abu-Ghazaleh, and Ryan Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *International Symposium on Microarchitecture*, 2014.

[11] Google. Confidential Computing. https://cloud.google.com/confidential-computing, 2022.

[12] Trusted Computing Group. Hardware requirements for a device identifier composition engine family 2.0, level 00, revision 78. https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-Device-Identifier-Composition-Engine-r78_For-Publication.pdf, 2018.

[13] Owen S. Hofmann, Sangman M Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[14] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G. Edward Suh. GuardNN: Secure DNN accelerator for privacy-preserving deep learning. In *Design Automation Conference*, 2022.

[15] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbah, and Emmet Witchel. Telekine: Secure computing with cloud GPUs. In *USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[16] Intel. Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.

[17] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[18] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, and et. al. In-datacenter performance analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture*, 2017.

[19] A. Khawaja, Landgraf. J, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *USENIX Symposium on Operating System Design and Implementation*, 2018.

[20] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[21] Jonathan M. McCune, Ning Qu, Yanlin Li, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[22] 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. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[23] ST Microelectronics. STM32 MCUs secure firmware install overview. https://www.st.com/resource/en/application_note/an4992-stm32-mcus-secure-firmware-install-sfi-overview-stmicroelectronics.pdf, 2022.

[24] ST Microelectronics. STM32H573 microcontroller with crypto accelerators. https://www.st.com/en/microcontrollers-microprocessors/stm32h743-753.html#documentation, 2022.

[25] NVIDIA. NVIDIA Confidential Computing. https://www.nvidia.com/en-in/data-center/solutions/confidential-computing/, 2022.

[26] Hyunyoung Oh, Kevin Nam, Seongil Jeon, Yeongpil Cho, and Yuneheung Paek. MeetGo: A trusted execution environment for remote applications on FPGA. *IEEEAccess*, 9:51313–51324, 2021.

[27] Emmanuel Owusu, Jorge Guajardo, Jonathan M. McCune, James Newsome, Adrian Perrig, and Amit Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *ACM Conference on Computer and Communications Security*, 2013.

[28] PCI-SIG. Integrity and Data Encryption (IDE). https://members.pcisig.com/wg/PCI-SIG/document/15149, 2022.

[29] PCI-SIG. TEE Device Interface Security Protocol (TDISP). https://members.pcisig.com/wg/PCI-SIG/document/18268, 2022.

[30] S. Pereira, D. Cerdeira, C. Rordigues, and S. Pinto. Towards a trusted execution environment via reconfigurable FPGA. In *ArXiv*, 2021.

[31] Mark Russinovich, Manuel Costa, Cedric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward confidential cloud computing. *Communications of ACM*, 64:54–61, 2021.

[32] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making trust between applications and operating systems configurable. In *USENIX Symposium on Operating System Design and Implementation*, 2006.

[33] Trusted Computing Group. DICE. https://trustedcomputinggroup.org/work-groups/dice-architectures/, 2022.

[34] Stavros Volos, Kapil Vaswani, and Rordrigo Bruno. Graviton: Trusted execution environments on GPUs. In *USENIX Symposium on Operating System Design and Implementation*, 2018.

[35] Samuel Weiser and Mario Werner. SGXIO: Generic trusted I/O Path for Intel SGX. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[36] Peichen Xie, Xuanle Ren, and Guangyu Sun. Customizing trusted AI accelerators for efficient privacy-preserving machine learning. In *ArXiv*, 2020.

[37] Miao Yu, Virgil D. Gligor, and Zongwei Zhou. Trusted display on untrusted commodity platforms. In *ACM Conference on Computer and Communications Security*, 2015.

[38] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. ShEF: Shielded enclaves for cloud FPGAs. In *International Conference on Architectural Support for Programing Languages and Operating Systems*, 2022.

[39] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Lixin Zhang, and Dan Meng. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *IEEE Symposium on Security and Privacy*, 2020.

A APPENDIX

A.1 Attack Vectors and Security Analysis

Table 1 summarizes the attack vectors discussed in Section 3 and, for those covered by our threat model, how ITX mitigates each of these attacks.

A.2 Firmware Provisioning and Device Certification

In this section, we describe an example process for firmware provisioning and device certificates that would be followed if ITX were to be used for IPUs in a production environment. During board manufacturing, the CCU would be provisioned with firmware followed by a board reset to harvest certificate signing requests (CSRs) generated by the execution of the primary and secondary bootloaders. The CSRs would then be used by the IPU manufacturer to issue device certificates.

Firmware Provisioning. The CCU is provisioned with firmware using the SoC's Secure Firmware Install (SFI) feature [23]. The firmware package consists of all firmware layers discussed in Section 5.1 and the configuration bytes (called `OPTION`), whose secure user memory registers are configured so that secure user memory includes only the regions onto which the secure bootloader is deployed. The firmware package is encrypted with a symmetric key, which is provisioned to a hardware security module (HSM).

The encrypted firmware package and the HSM are used by the board manufacturer to deploy CCU firmware during

Threat	Mitigation
Host (Software, Physical)	
IPU Memory Access, e.g., host software uses MMIO and PCI BARs, physical attacker tampers with on-chip memory	MMIO blacklist prevents CPU from accessing code/data in IPU; access via interfaces JTAG prohibited; IPU memory cannot be physically accessed w/o breaking into the package.
Host CPU, Memory, and PCIe bus, e.g., read/write, replay, or re-ordering of code/data in host memory or in transit, including DMA buffers, PCIe bus	Code/data are encrypted with AES-GCM with explicit IVs, and keys not shared with the host; uniqueness and integrity of IVs are ensured by trusted code executed on tiles.
IPU Binary Malleability, e.g., host replaces model encryption key or encrypted code	Bootloader computes hash of the tile binary; hash accumulated and checked against expected measurement in the job manifest. (Not implemented.)
IPU Connectivity, ICU-CCU or ICU-IPU Tampering on the development board	**no**; attacker can mount a physical attack to (1) retrieve the key(s) sent to IPU, and (2) tamper with ICU FW measurement sent to CCU
IPU-IPU Tampering	**no**; attacker can mount physical attacks against multi-IPU tasks; tamper with data sent across IPUs.
Supply Chain, Firmware	
Primary Bootloader Provisioning Tampering	The IPU manufacturer checks whether the signed bootloader manifest includes the expected nonce provisioned into the CCU primary bootloader.
Using non-genuine, known vulnerable TCB components	Firmware authorization; hardened measurement protocol outlined in Appendix A.4.
Side-channels	
IPU Memory	IPU memory access patterns cannot be observed by co-located attacker as the IPU is entirely assigned to one job at a time.
Host Memory and PCIe Bus	**no**: attacker can observe access patterns to host memory and on PCIe bus. However, these patterns do not leak much information in the BSP model, e.g., the size and number of minibatches, but not their contents.
Power and Timing	**no**: attacker can measure power consumption and/or execution time of a superstep. Similarly, this does not leak much information for typical ML tasks.

Table 1: Potential threats and how ITX mitigates them. Physical access attacks on the CCU-ICU-IPU and the IPU-IPU channels can be mitigated once the CCU is integrated on the IPU and AES-GCM is utilized to protect the IPU-IPU channels.

the manufacturing and testing. The chip tester implements a multi-stage protocol between the CCU secure bootloader and the HSM, during which the HSM authenticates the certificate issued by the CCU and wraps its firmware encryption key using the certified public key. This enables the CCU secure bootloader to decrypt the firmware package, to install the firmware, and to configure the `OPTION` bytes based on the requested configuration.

While this SFI process guarantees confidentiality of the firmware, it does not directly protect its integrity: provisioning may be subject to supply-chain attacks that would replace CCU parts provisioned using SFI with CCU parts containing malicious firmware. We extend SFI with protection against such attacks by injecting a secret known only to the IPU manufacturer into the primary bootloader. Once the CCU has been integrated onto an IPU board, a challenger can ask the primary bootloader to prove possession of the secret.

This process entails the following three steps. First, the IPU manufacturer generates a fresh secret for every batch of CCUs. The secret is injected to the primary bootloader of the CCU firmware. Second, the IPU manufacturer derives from the secret an asymmetric batch-specific bootloader manifest signing key. After deriving this key, the IPU manufacturer keeps only the public part. Third, the IPU manufacturer issues a certificate for the public bootloader manifest signing key. The certificate is signed by the IPU manufacturer Firmware certificate authority (CA). This certificate contains a batch number, and is valid till the production date of the batch.

Device Certification. In order to certify its device identity keys, the board tester resets the board and harvests the CIK and PIK CSRs generated for the board and platform identity keys, as well as the bootloader manifest. The command to harvest the bootloader manifest includes a fresh nonce, to be echoed in the signed bootloader manifest.

In response, the IPU manufacturer verifies the CSRs received by the card manufacturer and issues CIK and PIK certificates that are signed by the CIK and PIK CAs of the IPU manufacturer. In addition, the IPU manufacturer validates the bootloader manifest against the bootloader manifest signing key certificate specific to the batch to which the CCU belongs, and ensures that the nonce matches the expected one.

A.3 Firmware Updates

The CCU firmware includes a secondary bootloader and a CCE, both authenticated by the primary bootloader and possibly updated after the card has been deployed in production.

Updating the Secondary Bootloader. The secondary bootloader involves relatively complex cryptographic operations, and may need to be updated in the field. As discussed in Section 5, the platform identity key (PIK) is derived from UDS depending on the hash of the secondary bootloader. Therefore, any updates to the secondary bootloader changes the platform

identity, and PIK certificates issued by the manufacturer are no longer valid, requiring device re-certification.

Unfortunately, re-certification of a remote device by the IPU manufacturer can be a complex and lengthy operation as the manufacturer (by design) does not retain unique device keys. Thus, it requires collection of CSRs from the device, and more importantly an authentication mechanism to ensure that the manufacturer signs only PIK certificates exported from devices in the cloud provider's datacenters.

We overcome this challenge via a protocol that enables updates to the secondary bootloader without invalidating manufacturer-issued certificates.

Prior to updating the secondary bootloader (say to version Y), the cloud provider's IPU Firmware CA issues a TCB update certificate capturing the measurement of the new version of the secondary bootloader and revokes previous certificates for versions of the secondary bootloader that should no longer be deployed.

After a firmware update has been deployed, the primary bootloader generates a new CDI (CDI^Y). The secondary bootloader generates platform identity and attestation keys specific to this version of firmware (PIK^Y and AK^Y). However, the card identity key (CIK) stays the same as it does not depend on the measurement of the secondary bootloader. The PIK^Y certificate, hence, is signed by the original CIK, which has been certified by the manufacturer.

Subsequently, a remote challenger can combine the TCB update certificate with the CIK certificate originally issued by the manufacturer to verify the PIK^Y certificate is issued by the device using the original CIK, and that the measurement of the new secondary bootloader in the PIK^Y certificate matches the measurement of the secondary bootloader in the TCB update certificate.

Updating the CCE. Updates can be applied at any point without the need for any additional certification from the manufacturer. When a device boots with a new version of CCE, it generates a new attestation key with a signature over the public AK along with a hash of the CCE using the PIK. Quotes generated by the updated version of CCE firmware can be validated using a valid PIK certificate.

A.4 Measured Boot Protocol

The protocol discussed in Section 5.1 is still susceptible to advanced chosen-firmware attacks: a malicious secondary bootloader could impersonate another version of the firmware by using CIK to endorse a PIK certificate for the corresponding firmware measurement. Firmware authorization provides a strong defense against such attacks—the malicious firmware would need to be correctly signed by the IPU manufacturer to run as secondary bootloader. We can harden the boot protocol further by moving CIK and PIK generation into the primary bootloader (as shown in Figure 10) without revealing the private CIK to the secondary bootloader.

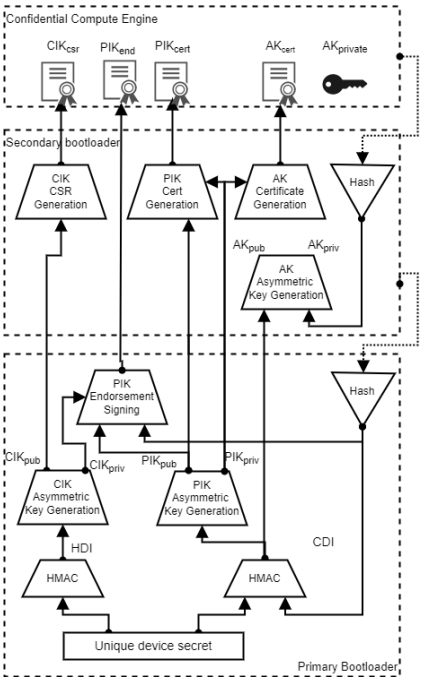


Figure 10: Hardened boot protocol that protects against bootloader impersonation attacks.

In this protocol, the primary bootloader generates CIK from UDS, and generates PIK using CDI and the measurement of the secondary bootloader. To allow a relying party (e.g., IPU manufacturer CA) to attest that the PIK was indeed generated by the primary bootloader, the primary bootloader creates a custom structure, known as PIK endorsement, containing the PIK public key along with a measurement of the secondary bootloader, and a signature over these two attributes using the CIK. The bootloader then scrubs the CIK private key and passes public CIK and PIK keys along with the private PIK and the PIK endorsement to secondary bootloader. During manufacturing, the IPU manufacturer PKI issues a PIK certificate only after validating the PIK endorsement structure. (Our prototype CCU does not implement this protocol to keep the primary bootloader simple.)

A.5 Compiled Manifests and Bootloader

Job Manifest. The compiler-generated job manifest includes all the information required by the IPU runtime and CCU to create and launch a new TEE, which will host the ML task. The manifest contains the hash digest of the application binary loaded into each IPU. It lists the synchronization points at which the IPU needs to synchronize with the host, and for each synchronization point, it keeps the following information:

- the key region identifier assigned to each stream that will be read or written following the end of synchronization (i.e., the mapping between a stream identifier j to a key region identifier);

- the ring buffer region (i.e., Tile PCI space in the ring buffer) assigned to each key region (key region definition registers);
- the part of each stream that has been mapped to the ring buffer region (stream offset);
- the set of physical key contexts to which the stream key needs to be loaded;
- the physical key context assigned to each exchange block context (exchange block context map registers); and
- the key region to which each physical key context is assigned (physical key map registers).

Secure Code Bootstrapping. The code snippet below illustrates the bootloader code that fetches the application binary frames and confirms the integrity of the IV of each frame.

```
def bootloader():
  IPU_id = get_current_ipu_id()
  tile_id = get_current_tile_id()
  num_frames = TOTAL_TILE_MEMORY / (MAX_FRAME_SIZE
       - IV_SIZE - TAG_SIZE)
  for index in range(1, num_frames):
    expected_iv = StreamType::CODE | ipu_id |
        tile_id | index
    frame = read_next_frame_from_host()
    if expected_iv != get_iv(frame):
      raise_security_exception()
    strip_iv_and_tag(frame)
  compute_hash()
```

A.6 Attestation

Cryptographic Operations. Table 2 details the keys sampled, derived, and exchanged at the start of a run in trusted mode. We rely on standard algorithms: Elliptic Curve Diffie-Hellman for establishing shared secrets, a KDF for deriving keys, and an AES-based authenticated key-wrapping scheme. These operations rely on the attestation of the manifest and runtime parameters, including all public keyshares. Each party provides its own random nonce, and the CCU combines them to deterministically derive keys for checkpoints and the final model; these keys are fresh secrets as long as one party is honest. To resume from a checkpoint saved in a previous run, the attested runtime parameters ensure that all parties agree on the epoch counter and checkpoint identifiers, and the parties provide their nonces for the previous and new run.

Remote Attestation. During TEE creation, the CCU generates an attestation report that captures security-critical attributes about the IPU and runtime configuration, including:

- the measurement of configuration registers;
- the measurement of the IPU bootloader used for loading application binaries onto IPUs;
- the measurement of the job manifest;

Key or secret		**Provider**	**CCU**
public/private keyshare for each relying party p	X_p, x_p	fresh EC share	receive X_p
encryption key for each input stream j	k_j	fresh key	unwrapped
public/private keyshare for the CCU in this run	Y, y	receive Y	fresh EC share
nonce for p in this run	$s_{p,Y}$	fresh secret	unwrapped
wrapping key for p, Y with salt $a = X^p \|\| Y \|\| M$	w_p	$\mathsf{KDF}[x_p \cdot Y](a)$	$\mathsf{KDF}[y \cdot X_p](a)$
key to load checkpoints saved by prior run Z	k_{load}	N/A	$\mathsf{KDF}[\vec{s}_{p,Z}]('ck')$
key to save checkpoints	k_{save}	N/A	$\mathsf{KDF}[\vec{s}_{p,Y}]('ck')$
key to save final model	k_m	unwrapped	$\mathsf{KDF}[\vec{s}_{p,Y}]('m')$

Table 2: Keying for a workload with manifest M between relying parties identified by their public keyshares $\vec{X}_p$ and a CCU identified by its fresh CCU public keyshare Y for this run. After attestation, an ECDH shared secret w_p is used for wrapping k_j, $s_{p,Y}$, and $s_{p,Z}$ when resuming from Z from p to the CCU, and optionally for wrapping k_m from CCU to any party p designated as a receiver of the final model. The keys used for encrypting checkpoints and the final model are derived from nonces from all relying parties, ensuring these keys are fresh (as long as one party is honest) and require agreement from all parties to be released.

- the hash digest of the attributes for this run, including:
  - the public keyshare of the CCU for this run (Y);
  - the epoch e and checkpoint counter c from which the job is restarted (if any);
  - the certificate fingerprints of all parties ($\vec{X}_p$);
  - a stream assignment, specifying a party for each input, and parties (model receivers) that receive the model key.

The host collects the attestation report, along with a CCU-issued certificate chain, which includes the AK, PIK and CIK certificates, and is rooted at the self-signed CIK certificate. These are presented to relying parties along with: the original CIK and PIK certificates, the TCB update certificates for the secondary bootloader and ICU firmware, and any intermediate CA certificates.

A relying party can verify the attestation report as follows:

1. Validate the CCU-issued certificate chain and auxiliary certificates; and check for certificate revocation.
2. Confirm that public key of the CIK certificate issued by IPU manufacturer matches the public key in the CIK certificate obtained from the CCU.
3. Confirm that any updates to the secondary bootloader and ICU firmware are rooted to a valid certificate chain. Two checks are required: (i) if there exists a TCB update certificate issued for the secondary bootloader with a

hash digest matching the hash digest in the CCU-issued PIK certificate; (ii) if there exists a TCB update certificate issued for ICU firmware with a hash digest matching the hash digest in the CCU-issued PIK certificate.

4. Review the attested manifest and attributes for this run.

Secure Key Exchange. For each run, each party p derives a fresh wrapping key w_p using its private keyshare x_p and the public keyshare of the attested CCU Y. This key is used to wrap a key package containing the streams identifiers assigned to the party and the party's key for these streams k_j, and the nonce(s) $s_{p,Y}$ for the current run (and $s_{p,Z}$ for the previous run if the current run is resuming from a checkpoint saved in run Z.) The CCU can derive the wrapping key for party p using its private keyshare y and the party's public keyshare X_p. In possession of $\vec{w}_p$, the attested CCU can unwrap the key packages of all parties, which are made available during the TEE launch stage.

The parameters of the model are encrypted using the final-model key k_m that has been derived by the CCU using the nonces obtained from all parties. The parties engage in a protocol for exchanging their nonces so they can derive the key once they possess all nonces. The CCU can additionally release the final-model key to model receivers listed in the attestation report using the wrapping key shared between itself and each model receiver.

A.7 Secure Checkpointing

Each IPU periodically checkpoints its state to enable recovery from failures. A checkpoint is created by writing the weights of the model to an output stream. The checkpoint also includes metadata, such as the current offset for all confidential data streams. These offsets are also written in plaintext, so that the IPU runtime can restart the job and resume loading of confidential data streams at the correct offset. Conversely, a checkpoint is restored by reading the weights using an input stream and resuming confidential streams from the checkpointed offsets. A checkpoint along with the job manifest and binaries suffice to resume an application from the checkpoint instead of restarting from the beginning.

In trusted mode, checkpoints are encrypted and integrity protected. In particular, tiles enforce the integrity of the process of restoring state from a previously created checkpoint. This includes protecting against attacks, such as tampering a checkpoint or loading a wrong checkpoint onto an IPU. (Guaranteeing freshness, e.g., resumption from the latest checkpoint, would involve some form of trusted persistent storage and is out of scope in this paper.)

Checkpoints are implemented using confidential streams. The code generated to read a checkpoint stream generates a sequence of expected IVs, checks that the IVs returned in the frames match the expected IV, and strips the IV and authentication tag from the frames. Conversely, the code generated to write a checkpoint stream generates a sequence of IVs and places them in the header of the frame. The IV for each frame uniquely encodes the checkpoint type, the epoch counter (incremented at each resumption), the checkpoint identifier (incremented at each saved checkpoint), the IPU and tile IDs, and the frame index. The CCU uses a separate key for each epoch; it installs the key of the epoch of the checkpoint it is resuming from, if any, and the key of the current epoch for writing all its checkpoints.

The tiles read and write checkpoints as follows:

1. Tiles obtain initial values of the epoch counter and checkpoint identifier (assigned by the CCU along with the bootloader code) from pre-determined locations in tile memory. If the epoch counter is not null, the tiles use it (with the checkpoint identifier) to compute their expected IVs and read part of their corresponding checkpoint.

2. Each tile increments their local epoch counter and start (or resume) the application.

3. At regular intervals, the tiles checkpoint their part of the state, using IVs computed from their current values, and then increment their local checkpoint identifier.

A.8 Sample Training Scenario

Figure 11 shows a sample training scenario with three parties. Given the job manifest generated by the compiler, IPU runtime, CCU, and IPU synchronize at various points where the IPU runtime populate the ring buffer with the data expected by the IPU, and the CCU loads keys to the IPU SXPs.

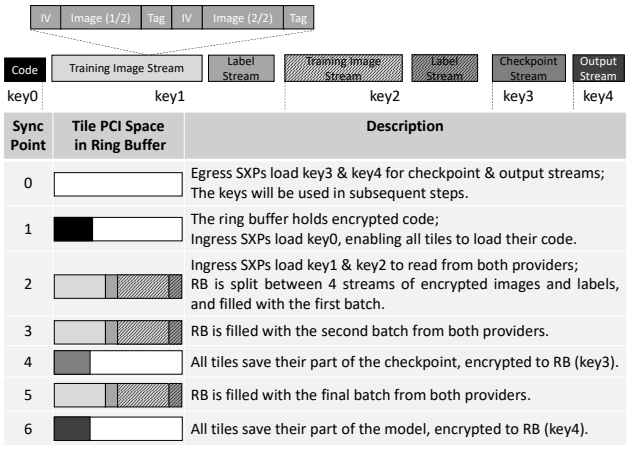


Figure 11: Sample training scenario with 3 parties: one providing model code (using key0) and the others (using key1 and key2) each providing their own streams of training images and labels; this task saves checkpoints (using key3) and a final model (using key4). The compiler emits a job manifest that indicates, for each synchronization point of the task, which part of each stream is mapped to the ring buffer (1..6) and which keys the CCU should load for ingress. The keys for egress streams are programmed in the start of the job (0).

Arbitor: A Numerically Accurate Hardware Emulation Tool for DNN Accelerators

Chenhao Jiang
University of Toronto
Vector Institute

Anand Jayarajan
University of Toronto
Vector Institute

Hao Lu
University of Toronto

Gennady Pekhimenko
University of Toronto
Vector Institute

Abstract

Recently there has been considerable attention on designing and developing hardware accelerators for deep neural network (DNN) training workloads. However, designing DNN accelerators is often challenging as many commonly used hardware optimization strategies can potentially impact the final accuracy of the models. In this work, we propose a hardware emulation tool called Arbitor for empirically evaluating DNN accelerator designs and accurately estimating their effects on DNN accuracy. Arbitor takes advantage of modern machine learning compilers to enable fast prototyping and numerically accurate emulation of common DNN optimizations like low-precision arithmetic, approximate computing, and sparsity-aware processing on general-purpose GPUs. Subsequently, we use Arbitor to conduct an extensive sensitivity study to understand the effects of these optimizations on popular models such as ResNet, Transformers, Recurrent-CNN, and GNNs. Based on our analysis, we observe that DNN models can tolerate arithmetic operations with much lower precision than the commonly used numerical formats support. We also demonstrate that piece-wise approximation is effective in handling complex non-linear operations in DNN models without affecting their accuracy. Finally, enforcing a high degree of structured sparsity in the parameters and gradients can significantly affect the accuracy of the models.

1 Introduction

Deep neural networks (DNNs) have shown unprecedented accuracy on many complex tasks like computer vision [26, 44], natural language processing [15, 65], recommendation systems [50], speech recognition [6], and robotics [55]. This superior performance of DNNs, however, comes at the expense of high computational costs in the training process. As the models are getting larger and more complex, the computational requirement for training these models has also been growing steadily. Therefore, designing and developing specialized hardware accelerators for DNN training has become an active area of research in both industry and academia [16].

Modern DNN accelerators are equipped with thousands of parallel processing units to leverage the abundant computational parallelism available in DNN training workloads. Moreover, they also employ many hardware-level optimizations that are designed to take advantage of the error-resilient nature of the DNN models. For example, DNN accelerators commonly use *low-precision numerical formats* for arithmetic operations to improve hardware utilization and energy efficiency [18, 48, 74]. Additionally, *approximate computing* techniques like the linear approximation of non-linear functions are widely used to reduce the hardware design complexity and cost [7, 43]. Finally, modern accelerators support *sparsity-aware processing* cores to minimize redundant computations in DNN workloads by skipping arithmetic operations over zeros. In recent years, there have been many studies proposing different variations of such hardware-level optimizations that have shown to be highly effective in improving the performance of DNN training workloads [56, 72].

Despite being a well-studied area, making the right design decisions for DNN accelerators is still a challenging task as many of the proposed hardware optimizations have non-trivial effects on the convergence accuracy of the DNN training algorithm and can potentially hurt the final accuracy of the model [18, 48]. Understanding the influence of these optimizations on the convergence of different DNN models requires rigorous experimental analysis on the DNN accelerator design. Unfortunately, currently available methods and tools are inadequate for such analysis as we explain below.

Software-based architectural simulators are widely used for the initial prototyping and analysis of different hardware design choices [8, 9, 34, 64]. These tools are primarily designed to perform hardware simulations at circuit-level precision and measure accurate low-level performance counters like instructions per cycle (IPC), cache miss rate, and branch prediction accuracy. As a result, architectural simulators are generally $6-7$ orders of magnitude slower compared to the real silicon performance [20, 63]. Since standard DNN benchmark models [49, 75] take anywhere from hours to months of training to reach peak accuracy, using software-based simulators to

analyze DNN accelerator designs can get prohibitively time-consuming. Even though other approaches like FPGA-based prototyping can offer performance close to the real hardware, programming FPGAs requires unique expertise that is rare even among hardware researchers. Moreover, building the software stack that can provide runtime support for training DNNs on FPGAs requires substantial engineering effort.

To address the limitations of these traditional methods, recent works [45, 47, 59, 70] have proposed hardware emulation frameworks specifically designed for analyzing the accuracy effects of common DNN optimizations like low-precision training. These tools provide convenient APIs to configure and emulate low-precision arithmetic in the training process by modifying the computation graphs of DNN models. Unlike architectural simulators, DNN emulators are built as extensions over popular machine learning frameworks like TensorFlow [3] or PyTorch [54] and can be executed over general-purpose accelerators like GPUs. This enables fast prototyping of arbitrary low-precision numerical formats and evaluating their effects on standard DNN benchmark models within reasonable time frames. However, we observe that extending these tools to support other hardware features like approximate or sparsity-aware computing require significant effort from the user. On top of this, current DNN emulators are designed to perform low-precision arithmetic emulation at the DNN layer-level granularity. Under this strategy, the computation of individual layers in the DNNs like matrix multiplication and convolution is performed using standard 32-bit floating point format, and the output is rounded down to the user-defined numerical format. We observe that this *coarse-grained emulation* approach can produce numerically inaccurate results compared to the real hardware. Our empirical evaluation reveals that the relative numerical error of low-precision arithmetic emulation in state-of-the-art DNN emulators can be as high as 15% (more details in section 3). We argue that such inaccurate emulations could lead to incorrect evaluation of hardware designs in DNN accelerators.

In this work, we build an easy-to-use, extensible, and more importantly numerically accurate emulation tool called Arbitor for empirically evaluating DNN accelerator designs. Arbitor is built on top of TensorFlow and provides extensions to its Keras front-end APIs for users to easily configure and emulate common hardware features like low-precision training, approximate computing, and sparsity-aware processing on standard DNN models. In contrast to other DNN emulators, Arbitor emulates the user-defined hardware features at the granularity of the primitive mathematical operations in the DNN layers to accurately mimic the behavior of real hardware. We support this *fine-grained emulation* in Arbitor with the help of the XLA compiler-backend in TensorFlow [23]. Modern machine learning compilers like XLA use domain-specific intermediate representations (IR) for defining the operations in the DNN computation graphs. These IR definitions of DNN operations are designed to be hardware-independent and are progressively lowered to GPU executable kernels by the compiler. We make a key observation that it is possible to automatically generate GPU kernels that emulate the user-defined hardware features by modifying the code generation pipeline of TensorFlow XLA. We empirically show that, compared to the state-of-the-art layer-level emulation approach, the operator-level emulation strategy of Arbitor can perform arithmetic operations and generate results that accurately match the results from real hardware.

We subsequently use Arbitor to conduct a series of sensitivity studies on the effects of low-precision training, approximate, and sparse computing on popular DNN architectures like Transformer [65], ResNet-18 [26], Convolutional Recurrent Neural Network (CRNN) [66], and Graph Neural Network (GNN) [73]. First, we conduct an extensive analysis of various non-standard floating point specifications and find that DNN models can maintain their accuracy with much lower precision than many standard floating point formats supported in current DNN accelerators. We also observe that the numerical precision can be reduced even further by combing low-precision formats with the standard single-precision floating point. Second, we also analyze the effectiveness of newly proposed non-floating point numerical formats like Posit [25] on DNN training. We find that, contrary to the prior observations [59], Posit does not yield better model accuracy compared to floating point. Third, we evaluate a popular approximate computing optimization technique called piece-wise linear approximation [35] and find that natural language processing (NLP) models like CRNN can maintain their baseline accuracy even with aggressive approximations. Finally, we analyze the effect of sparse computing and observe that enforcing more than 50% sparsity on DNN training can significantly affect the model accuracy. To the best of our knowledge, we are the first open study to conduct such an extensive analysis of common DNN accelerator designs using numerically accurate methods and tools.

In summary, we make the following contributions:

- We highlight that the hardware research community is currently lacking a fast and accurate hardware analysis tool for DNN training accelerators as state-of-the-art tools are either prohibitively slow or are susceptible to numerical inaccuracy.
- We build a hardware emulation tool Arbitor using a compiler-assisted fine-grained emulation strategy for numerically accurate emulation of optimizations like low-precision, approximate, and sparse computation.
- Using Arbitor, we conduct the first in-depth empirical analysis on the effects of low-precision arithmetic, approximate computing, and sparsity-aware processing on the accuracy of popular DNN models like Transformer, ResNet-18, CRNN, and GNN. We will be open-sourcing Arbitor soon for supporting empirical research in DNN accelerators.

2 Hardware Accelerators for DNN Training

Modern DNN models [15, 17, 26, 44] contain millions to even trillions of parameters and are trained for hours to months by iteratively processing large batches of data from humongous datasets and updating the model parameters to minimize the prediction error until the model converges to the desired accuracy. The DNN training process primarily consists of the repeated execution of computationally expensive algebraic functions such as matrix multiplication, convolution, and element-wise operations. Fortunately, these functions exhibit abundant computational parallelism which can be leveraged to speed up the training process using parallel processing hardware accelerators like graphics processing units (GPUs).

While GPUs have been the mainstay for DNN training, in recent years, there has been an increasing interest in developing more specialized accelerators. For example, Google TPU [31], Habana Gaudi [30], Graphcore IPU [24], Cerebras Wafer-Scale engine [14], and Amazon Trainium [5] are a few notable examples of commercial DNN accelerators. In addition to massively parallel processing capabilities, these accelerators also employ several hardware-level optimizations that exploit the inherent error-resilient nature of DNNs to improve training performance and hardware utilization. Below, we describe the three most common categories of hardware-level optimizations supported in DNN accelerators.

1. Low-precision arithmetic. Unlike regular parallel processing applications that require high-precision floating point computation, DNN models are highly tolerant towards using reduced-precision arithmetic due to the error-correcting nature of the training algorithm. Therefore, many DNN accelerators support low-precision numerical formats that use fewer bits than the standard 32-bit single-precision floating point (FP32). This enables accelerators to provide higher processing power from the same hardware budget. For example, Nvidia GPUs can perform 2× higher floating point operations (FLOPS) with half-precision (FP16) than using single-precision format [52]. Additionally, low-precision data formats can also help reduce the memory footprint of DNNs and in turn, improve the cache utilization and lower the memory bandwidth pressure during training. As a result, many modern DNN accelerators support a wide range of low-precision floating point formats outside the traditional IEEE-754 standard [2] as shown in Figure 1.

In general, floating points are represented using a sign bit, exponent bits, and mantissa bits. The range and the precision of a particular format are determined by the number of exponent and mantissa bits respectively. Therefore, different low-precision formats make the fundamental trade-off between the range and the precision of values the type can represent. For example, FP16 uses 5 exponent and 10 mantissa bits and has a limited range (i.e., $\pm 65,504$) compared to FP32 (i.e., $\pm 3.40 \times 10^{38}$). An alternative 16-bit format called brain float 16 (BF16) [32], on the other hand, uses 8 exponent bits and 7 mantissa bits and can support a similar range of values as that of FP32. Since BF16 format trades precision in favor of a wider range, it has shown to be better suited for DNN training than the standard FP16 and is widely supported in many DNN accelerators [32].

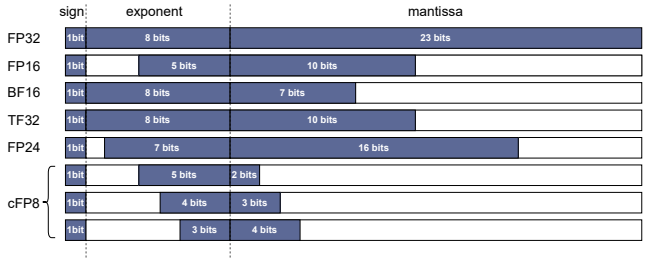


Figure 1: Common floating point formats in DNN accelerators

In addition to floating point formats, hardware researchers have also been exploring other data formats e.g., fixed-point arithmetic [41, 71, 74]. In contrast to floating points, fixed-point arithmetic lends itself to a simpler hardware design with a smaller chip area and lower power consumption [27] but lacks the dynamic range and precision that floating point formats have. In recent years, there has also been proposals [13, 46] in using the novel Posit formats [25] for DNN training as it takes less circuitry than floating point processing units while providing a wider dynamic range. Despite this significant research attention, finding numerical formats that make the right trade-offs in DNN training workloads is still an open research problem.

2. Approximate computation. The training computation of DNN models is mostly dominated by linear algebraic functions like matrix multiplication and convolution. These functions are composed of numerous primitive mathematical operations like addition and multiplication that maps very well with the thousands of parallel arithmetic units provided by the DNN accelerators. In addition to these, many DNN models also contain operations that use non-linear functions. For example, modern NLP and image classification models use complex activation functions like tanh, sigmoid, GeLU [28], and Swish [58] which are shown to play an important role in achieving high accuracy for the models. However, the exact computation of these functions is often very expensive to perform in the hardware because of the exponentiation and division terms present in the functions. Instead, DNN accelerators employ approximations of such functions that are cheaper to implement in hardware. Examples of such approaches include piece-wise linear/non-linear approximations [35, 61], lookup table [37], bit-level mapping [69], or a hybrid of these methods. Such approximations in DNN accelerators enable them to achieve improved hardware performance and lower chip area at the expense of imprecise computation of the non-linear functions. Therefore, more aggressive approximations can potentially affect the convergence of DNN training.

3. Sparsity-aware computation. Large DNN models are known to exhibit a significant amount of redundancy due to the over-parameterization of the model architecture [22]. There have been several efforts in exploiting this redundancy to improve performance and reduce the memory footprint of DNN workloads. For example, software-level optimizations such as DNN pruning [10] has found effective in significantly reducing the model size by removing the redundant parameters from the model. However, naïvely applying DNN pruning often ends up with models having randomly distributed sparsity patterns. Since DNN accelerators are primarily designed to process dense data structures, extracting performance benefits from DNNs with unstructured sparsity is often challenging. To address this issue, modern DNN accelerators are equipped with specialized processing cores that can improve the performance of DNN applications in the presence of semi-structured sparsity patterns. For example, the latest Nvidia A100 GPUs support Sparse Tensor Cores that can accelerate matrix multiplication operations by 2× for operands having 2:4 structured sparsity.[1] Recent works [56, 72] have proposed hardware-software co-optimizations that follow structured DNN pruning strategies during the training process to take advantage of such sparsity-aware processing units. Despite the potential performance improvements, enforcing structured sparsity during training is shown to have a significant impact on the convergence of the DNNs [68]. Finding the right balance between structural sparsity and model accuracy is currently an active area of research.

3 Need for an Accurate Hardware Analysis Tool for DNN Accelerator Design

Many of the aforementioned optimization strategies have the potential to significantly improve the performance of DNN training. However, at the same time, depending on the degree and aggressiveness of the optimizations, they can also negatively affect the model accuracy. Therefore, designing accelerators for DNN training requires taking both hardware performance and accuracy effects of the optimizations into consideration. Even though there are standard methods and tools available to prototype different chip designs and estimate their performance, they fail to be a good fit for analyzing their effects on model accuracy as we explain below.

Limitations of Traditional Verification Tools. One of the most common and cost-effective approaches for prototyping and verifying hardware designs is to use architectural simulators [4, 29]. Hardware researchers use simulators like GPGPU-sim [8], gem5 [9], and Multi2Sim [64] to evaluate architectural design choices by running software-based simulations of the proposed features and collecting hardware performance counters like instructions per cycle (IPC), cache utilization, and energy consumption. These tools are designed to run simulations with circuit-level precision, but they come at the cost of high execution time. Our empirical evaluation shows that running a single 1024×1024 FP32 matrix multiplication using GPGPU-sim is more than 5 orders of magnitude slower than running on bare-metal GPUs. Since training involves iteratively executing many such operations and can take anywhere from hours to months to finish even on powerful hardware [49, 75], it becomes prohibitively time-consuming to use these simulators to analyze the convergence effects of hardware optimizations on modern DNN models.

FPGA-based prototyping is another approach followed in the industry for hardware design verification. Programmable chips like FPGAs allow accurate verification of the functional logic of a design and the ability to run benchmark applications on custom-designed chips at a speed closer to the performance of the physical hardware. But programming FPGAs to reliably implement the desired hardware features is a labor-intensive task and requires special expertise and infrastructure support that is often beyond the reach of many independent researchers [60]. On top of this, building a full-fledged software stack that can provide the requisite runtime support for training DNN models is a major engineering undertaking that requires significant time and financial investment.

Since analyzing the statistical effects of DNN accelerator optimizations using traditional hardware verification tools is difficult and time-consuming in practice, hardware researchers often end up making design decisions using limited empirical analysis of the design or based on speculations. As hardware manufacturing is an extremely lengthy and expensive process, this could potentially cost a substantial amount of time, money, and resources. To take a real-world example, Nvidia first introduced half-precision floating point (FP16) support in Tesla P100 GPUs [52] in 2016 to deliver higher performance for deep learning workloads. However, it was soon observed that using half-precision in training can cause serious convergence issues on many models as algebraic functions like matrix multiplication, convolution, and batch normalization perform reduction over large dimensions of matrices and can suffer from higher accumulation error compared to single-precision (FP32) training. To correct this issue, it took researchers another two years to find a software-level fix called mixed-precision training [48] that uses a combination of FP16 and FP32 precision (for computation and reduction respectively) to curtail the numerical errors. More recently, Nvidia introduced a 19-bit floating point format called Tensor Float (TF32) in their Ampere architecture-based GPUs [51] as a drop-in replacement for FP32 data type. This again caused major pushback from the machine learning community for the numerical instability it caused on certain non-standard deep learning workloads [57]. These anecdotes suggest the importance of having rigorous methods and tools for analyzing the accuracy effects of DNN accelerator designs as early in the hardware manufacturing process as possible.

[1] Here, an N:M sparsity indicates that out of every block of M contiguous values in the input operands, only N values are non-zero.

Inadequacy of Current Emulation Tools. To meet this unique requirement of DNN accelerator research and development, recent works have proposed hardware emulation tools such as TensorQuant [45], QPyTorch [70], and GoldenEye [47]. Unlike architectural simulators that perform expensive cycle-accurate simulations of the entire hardware, these tools are specifically designed for analyzing DNN optimizations like low-precision training by emulating arbitrary numerical formats on general-purpose accelerators like GPUs. State-of-the-art DNN emulators are built on top of popular machine learning (ML) frameworks like TensorFlow [3] and PyTorch [54], and provide extensions to their front-end APIs to configure and train DNN models using user-defined numerical formats. These tools take advantage of the fact that the ML frameworks represent DNN models as layers of well-defined algebraic functions, and therefore, the numerical errors introduced by low-precision arithmetic in DNN training can be emulated by replacing the individual layers with corresponding software-emulated functions. Since these tools are well-integrated with ML frameworks and allow fast experimentation through GPU-accelerated emulation, they make a convenient tool for quick exploration of the low-precision data formats on a wide range of DNN models.

Despite being a promising approach, current DNN emulation tools suffer from two major limitations. First, they are primarily designed to target only a *narrow scope* of DNN optimizations, namely, low-precision training. Supporting other optimizations like approximate or sparsity-aware computation require intrusive changes in these tools due to their tightly coupled design and implementation with the underlying ML frameworks and the targeting GPU backend. Second and more importantly, we observe that the current DNN emulators are susceptible to *numerical inaccuracies* due to a coarse-grained emulation strategy that they follow. Under this strategy, the low-precision arithmetic emulation is achieved by performing individual algebraic functions in the DNN model using the standard FP32 format supported in GPUs and then rounding the output down to the user-defined low-precision format. Even though this layer-level rounding approach lends itself to a simpler DNN emulator design, we observe that this strategy fails to accurately reproduce numerical errors that can occur with low-precision arithmetic. For instance, many DNN layers like matrix multiplication and convolution internally perform several primitive mathematical operations like multiplication and addition. Under the coarse-grained layer-level emulation approach, these primitive operations are performed using higher-precision FP32 arithmetic. As a result, they fail to account for the low-precision multiplication and accumulation error occurring within the algebraic functions.

$$rnd((a_1 * b_1) + (a_2 * b_2) + (a_3 * b_3)) \quad (1)$$

To illustrate this limitation, we use a simple matrix multiplication between a 1×3 matrix ($[a_1 \quad a_2 \quad a_3]$) and a 3×1 matrix ($[b_1 \quad b_2 \quad b_3]^T$) as an example. Equation 1 shows the

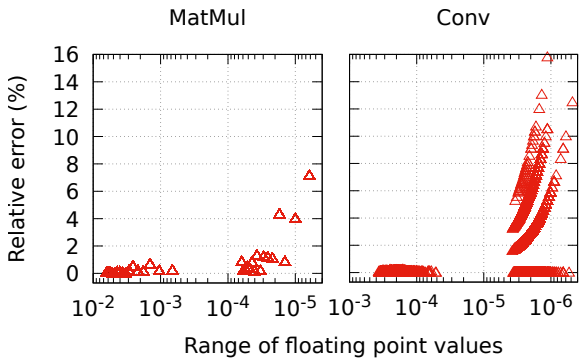


Figure 2: The numerical difference between real hardware and the layer-level emulation

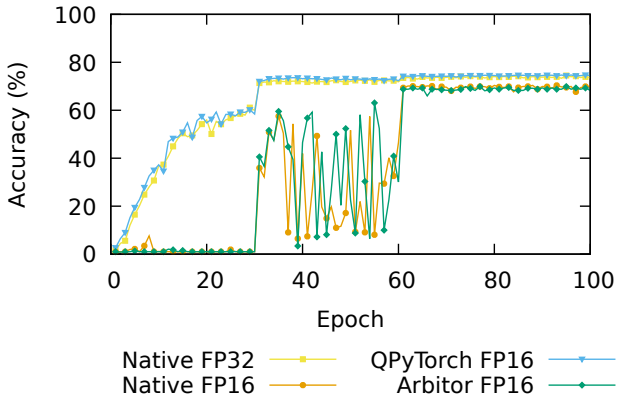


Figure 3: Comparison of training curves for ResNet-50 with native FP32, native FP16, QPyTorch-emulated FP16, and Arbitor-emulated FP16

definition of this matrix multiplication performed using FP16 arithmetic following the layer-level emulation strategy. In this case, all the primitive multiplication and addition operations are performed using FP32 arithmetic, and only the final result is rounded to FP16 using $rnd()$. In contrast, an accurate emulation of FP16 matrix multiplication requires rounding on every primitive operation as shown in Equation 2.

$$rnd(rnd(rnd(a_1 * b_1) + rnd(a_2 * b_2)) + rnd(a_3 * b_3)) \quad (2)$$

To empirically estimate the extent of the inaccuracy, we emulate FP16 arithmetic using the layer-level strategy on matrix multiplication and 3×3 convolution operation over 128×128 matrices. Then we compare the output matrix generated with that obtained using the native FP16 supported in real hardware such as Nvidia 2080 Ti GPU [53]. Figure 2 shows the relative error between the emulated and the hardware native outputs on different input matrices with floating point values randomly generated from the range $[10^{-6}, 10^{-2}]$. As we show, in comparison to the GPU native FP16 results, the re-

sult generated by the emulated version can differ by as much as 15.74%. Since DNN training is an iterative process, such numerical inaccuracies in emulation can accumulate over the course of training, leading to making incorrect assessments about the effects of low-precision data formats on the DNN accuracy. To demonstrate this, we first train ResNet-50 using both FP32 and FP16 on Nvidia 2080 Ti and get validation accuracy curves as shown in Figure 3. The highest accuracy with FP16 is 70.36%, which is 3.7% lower than FP32 due to the numerical error with low-precision arithmetic. We subsequently use the state-of-the-art QPyTorch which applies layer-level emulation to run the training with emulated FP16. However, the validation accuracy using emulated FP16 in QPyTorch is observed to be close to the native FP32 accuracy. This shows that the layer-level emulation strategy fails accurately reproduce the numerical error of FP16 arithmetic.

Based on the above observations, we argue that current DNN emulators are ill-suited for accurately estimating the accuracy effects of low-precision arithmetic on DNN training due to their inherent design limitations.

4 Arbitor: Overview

To address the aforementioned limitations, we propose Arbitor. Arbitor strives to achieve three main goals. First, to propose a DNN emulator with an *extensible design* that can emulate common hardware-level optimizations like low-precision training, approximate computation, and sparsity-aware processing. Second, to enable *easy prototyping* of these optimizations and facilitate empirical analysis on their statistical effects on popular DNN models. Finally, we strive to provide *numerically accurate emulation* support that precisely mimics the functional logic of the hardware optimizations on general-purpose accelerators like GPUs. To achieve these goals, we make the following implementation choices.

First, we build Arbitor on top of the popular machine learning framework TensorFlow [3]. TensorFlow supports the convenient and user-friendly Keras front-end APIs for writing DNN training applications and offers implementations of a wide range of state-of-the-art DNN models [33]. To allow fast prototyping of low-precision arithmetic, approximate computation, and sparsity-aware processing on DNN models, Arbitor provides extensions in the Keras APIs for the users to define two emulation policies: (i) Data type policy for defining arbitrary precision numerical formats and the implementations of basic primitive mathematical operations on top of the user-defined data types. (ii) Masking policy for enforcing $N:M$ sparsity patterns on the parameter updates during training.

Providing numerically accurate emulation support for these hardware features requires fine-grained manipulations of the computations involved in DNN training workloads. However, manual modifications to the implementation of algebraic functions in the ML framework are intrusive and require significant engineering effort. Such modifications can also hamper the extensibility of the emulation tool, as supporting new DNN models or hardware features would require potential code changes. To address these challenges, we make a key observation: despite the apparent differences among hardware features like low-precision arithmetic, approximate computing, and sparsity-aware processing, all of them can be emulated by manipulating a small set of primitive operations, such as addition, multiplication, memory read and write, which constitute the DNN algebraic function implementations.

Based on this observation, we design Arbitor with the help of the XLA compiler in TensorFlow [23] to provide fine-grained emulation support. The XLA compiler operates on an intermediate representation (IR) named HLO IR, which precisely represents the computation through a concise set of fundamental primitive operations. This representation format of DNN computation is particularly well-suited for fine-grained manipulations to support numerically accurate emulation. Moreover, due to the model and hardware-independent nature of HLO IR, this compiler-based design of Arbitor provides seamless support for a wide range of DNN models and easy extensions for the emulation of more DNN accelerator optimizations. The XLA compiler compiles the high-level computation graph of DNN progressively down to hardware-specific kernels using standard code generation techniques, as illustrated in Figure 5. Initially, the computation graph described in the Keras front-end is transformed to HLO IR. The XLA compiler takes the HLO IR as the input and proceeds to perform a series of optimizations and analyses, ultimately lowering the HLO IR to hardware-specific kernel implementations. Notably, XLA leverages LLVM infrastructure for generating kernels on GPU, which is the target environment of Arbitor. During this process, Arbitor incorporates user-defined emulation policies into the computation by modifying the compilation pipeline.

Below, we provide details about the emulation policies and the fine-grained emulation strategy supported in Arbitor.

4.1 Emulation Policies

Figure 5 shows an example of Keras computation using the two emulation policies supported in Arbitor. Users can define and configure the `dtype` and `mask` policies using the Keras APIs. The policies can be assigned either to specific layers (in the Figure 5), or as a global policy for the entire DNN model.

The data type policy allows users to configure low-precision arithmetic and approximate computing emulation. Under this policy, the user can define a custom data type specification as an implementation of an abstract C++ class called `Cus`. The specification should include two components. First, two **casting functions** to convert the custom data type to and from the standard FP32 data type. Second, implementations of **primitive mathematical operations** over the custom data type such as addition and multiplication, and mathematical functions such as exponent and logarithm.

```
class EmuBF16: public Cus {
public:
    unsigned int v;
    EmuBF16(unsigned int v) : v(v) {}
};
EmuBF16 from_float(float f) {
    unsigned int bits = *(unsigned int*)&f;
    unsigned int rounding_bias = 0x7fff + ((bits >> 16) & 1)
    return EmuBF16((bits + roounding_bias) >> 16);
}
float to_float(EmuBF16 b) {
    unsigned int bits = b.v << 16;
    return *(float*)&tmp;
}
EmuBF16 operator+(const EmuBF16& a, const EmuBF16& b) {
    return from_float(to_float(a) + to_float(b));
}
EmuBF16 operator*(const EmuBF16& a, const EmuBF16& b) {
    return from_float(to_float(a) * to_float(b));
}
....
```

Figure 4: Custom data type specification of BF16

Figure 4 shows a specification of BF16 [32] data type in Arbitor named `EmuBF16`. In this example, `from_float` and `to_float` are the two casting functions. In addition, the Figure also shows example implementations of addition and multiplication operations defined over `EmuBF16`. It should be noted that the abstract class `Cus` makes very few assumptions about the specification of the data type and the implementations of the primitive operators. This allows Arbitor to support a wide range of arbitrary numerical formats and customized operator implementations on top of them to emulate low-precision arithmetic and approximate computing.

Next, the masking policy allows the users to configure an arbitrary $N : M$ sparsity pattern on the layers during the training process. In addition to N and M, the masking policy also takes a scoring function as a configuration parameter. The scoring function is a user-defined function that takes an array of M values as input and assigns an importance score for each value in the array. The masking policy defined in Figure 5 uses the absolute value as the scoring function. Based on these configurations, Arbitor dynamically generates masks for each parameter and gradient matrices accessed during the training and selects the top N values with the highest importance score on every block of M contiguous values in the matrix.

Once the policies are defined and assigned, Arbitor automatically takes care of emulating the corresponding hardware feature in the DNN training computation as we explain below.

4.2 Compiler-Aided Fine-Grained Emulation

Arbitor generates customized GPU kernel implementations to achieve numerically accurate emulation of user-defined data types and masking policies by leveraging the XLA compiler. For the emulation of low-precision arithmetic and approximate computing, Arbitor injects user-defined data types and operator specifications into the generated kernels. This is accomplished by compiler passes that replace the primitive operations in the HLO IR with the corresponding user-defined operations and automatically insert casting operations for FP32 inputs, as shown in Figure 5. For example, when emulating the user-defined `EmuBF16` arithmetic in matrix multiplication, the default FP32 addition and multiplication operations within the kernel are replaced with the corresponding functions defined in the data type specification. This approach could also support emulating approximation of intricate mathematical operations, such as exponential function and hyperbolic tangent (tanh) function, where users are granted the capability to define customized implementations of these complex operations based on approximation techniques like the piecewise-linear approximation [35]. All of these are made possible because Arbitor makes few assumptions of specifics of data type representation and operator behavior that users provide.

Similarly, the masking policy in Arbitor enforces sparsity patterns on weight and gradient matrices by overriding the memory access operations to these matrices with masking operations, as shown in Figure 5. Therefore, whenever the values in the matrices are read during the training process, Arbitor dynamically generates the corresponding mask according to users' specifications and computes the element-wise product between the user-defined mask and the accessed matrix, returning the resulting masked values. This approach allows Arbitor to preserve the original matrices to handle dynamically changing sparsity patterns during training.

Emulating at the granularity of these primitive operations allows Arbitor to accurately mimic the numerical behaviors of the algebraic functions on real hardware, as shown in Equation 2. To empirically validate this, we re-run the experiment in Figure 2 and Figure 3 using Arbitor-emulated FP16 and compare the results against the hardware native FP16 results. The results of matrix multiplication and convolution using Arbitor's emulated FP16 precisely match with the hardware-native FP16 results with zero relative error, affirming that Arbitor is numerically accurate. Furthermore, we also show in Figure 3 that the validation accuracy curve of ResNet-50, employing Arbitor's emulated FP16 arithmetic, closely matches that obtained with native FP16. The margin of error in the validation accuracy achieved is only 0.48%. In contrast, the accuracy obtained by QPyTorch exhibits a discrepancy of 4.32% from native FP16 accuracy. These findings underscore Arbitor's capability to accurately estimate the effects of low-precision arithmetic on DNN training. In addition, we also analyse the emulation overhead of Arbitor compared to hardware native performance that can be found in Appendix B.

5 Arbitor: Case Studies

We build Arbitor to provide a reliable analysis tool for hardware research to explore DNN accelerator design space. We

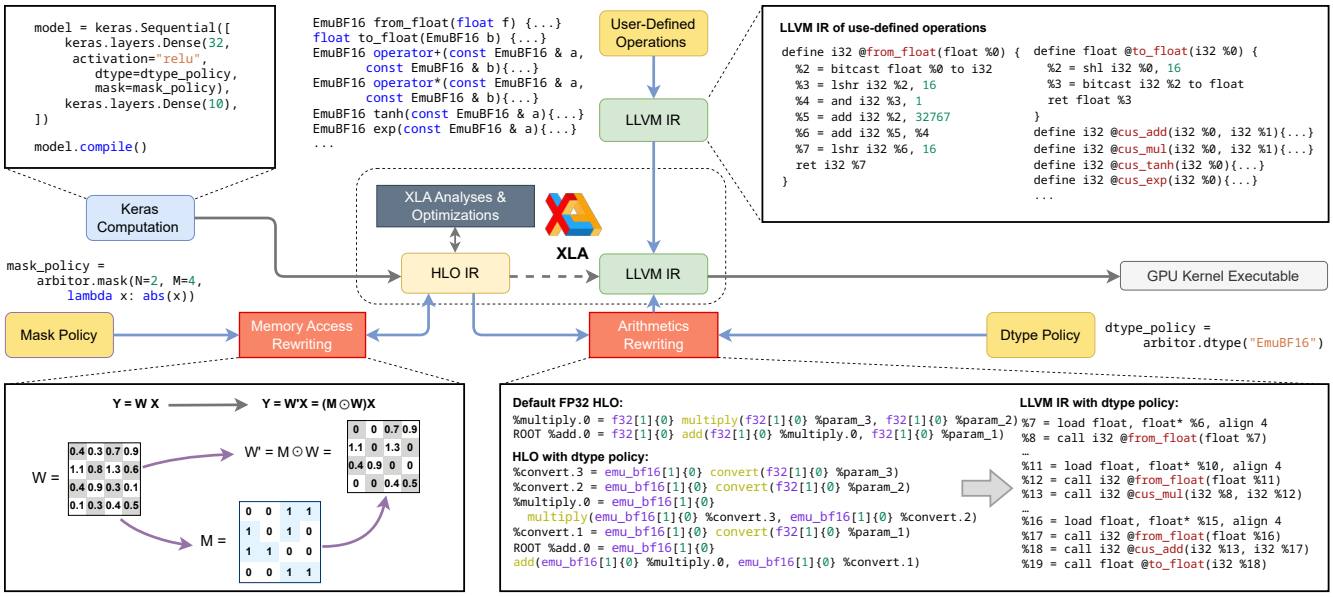


Figure 5: Arbitor workflow and multiple layers of IR

illustrate the benefits of Arbitor through a series of case studies on common hardware-level optimizations proposed for DNN accelerators. Specifically, we conduct an in-depth empirical analysis on the sensitivity of low-precision training, approximate computing, and sparsity-aware processing on DNN training. To the best of our knowledge, we are the first open study to conduct a numerically accurate and extensive study on these optimizations. We would also like to highlight that the case studies presented in this section are just a few examples of the many potential applications of Arbitor. We believe Arbitor is flexible and extensible to provide emulation support for an even wider range of hardware features.

5.1 Experiment Setup

Model	Dataset	Accuracy
ResNet-18 [26]	CIFAR-10 [36]	93.78
GNN [73]	Cora [62]	84.06
Transformer [65]	FordA [1]	87.24
CRNN [66]	eng-fra [12]	87.25

Table 1: Benchmark models, datasets, and baseline accuracy

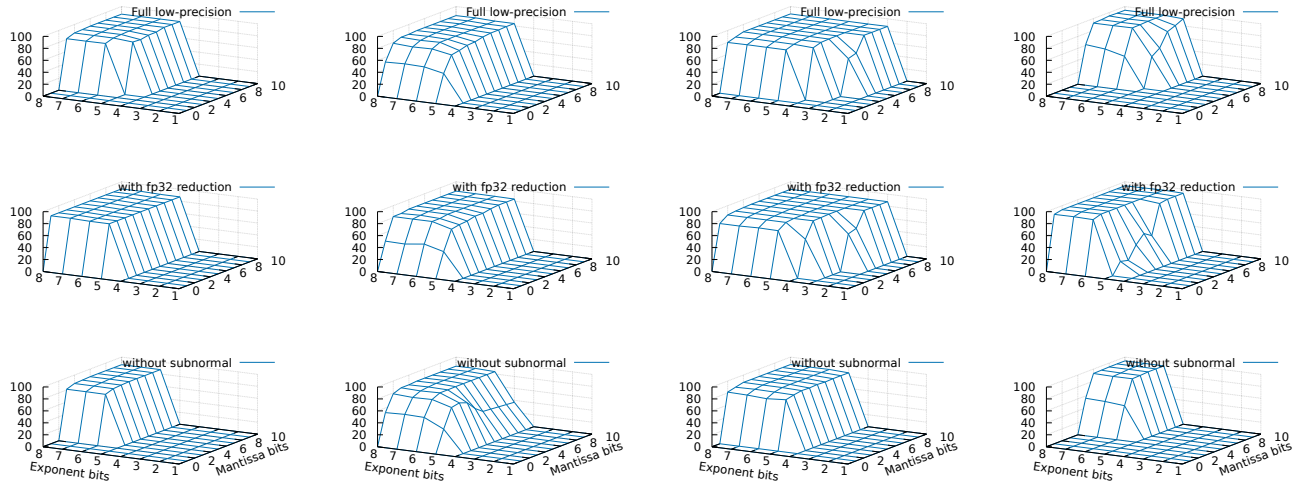
Workloads: Table 1 shows the DNN models we use in our experimental study, namely ResNet-18 [26], Transformer [65], GNN [73], and CRNN [66], selected from the Keras model hub [33]. These model implementations are based on a diverse range of model architectures that are part of the standard DNN training benchmark suite MLPerf [49]. For instance, ResNet-18 is a convolutional neural network (CNN) that is used for image classification applications. Transformer model is an attention-based DNN used in sequence classification and translation. GNN is a graph neural network used for node prediction in graph datasets. CRNN is a recurrent neural network (RNN) based model used for natural language translation.

Hardware and Runtime: We conduct our experiments on 32-core AMD EPYC 7371 machines with four Nvidia 2080Ti GPUs each with 12 GB memory. The runtime environment uses Ubuntu 20.04 with CUDA 11.0, cuDNN 8.0, and CUTLASS 2.6. Arbitor is built on top of TensorFlow v2.4.0 [3].

Metrics: In our experiments, we use the peak validation accuracy as the main evaluation metric. To measure the accuracy, we train ResNet-18 for 100 epochs, Transformer for 120 epochs, GNN for 300 epochs, and CRNN for 100 epochs on their respective data sets. We train each model three times and use their averaged accuracy for comparison to minimize the effects of slight variations in the final accuracy. We use the validation accuracy of each model trained using single-precision floating point (FP32) as the baseline for all our comparisons. The baseline accuracy of each model is shown in Table 1.

5.2 Case Study #1: Low-Precision Training

We use Arbitor to investigate the impact of low-precision arithmetic on the model accuracy by training the DNN models in Table 1 with different numerical formats. Current DNN accelerators support a variety of floating point formats with different numbers of exponent and mantissa bits, as described in Section 2. Therefore, we conduct a sensitivity study on different floating point formats and analyze how they affect the model accuracy. In addition, we also evaluate the effects of other numerical formats like Posit [25] on DNN training.



(a) CRNN (b) GNN (c) Transformer (d) ResNet-18

Figure 6: Validation accuracy of models trained using different exponent and mantissa widths of floating point format

5.2.1 Sensitivity Study on Floating Point Numbers

Single-precision floating point (FP32) is one of the most commonly used numerical formats for DNN training. According to IEEE 754 standard [2], FP32 is specified as follows.

$$value = \begin{cases} (-1)^s \times 2^{-126} \times 0.m & \text{if} \quad e = 0x00 \quad \text{(subnormal)} \\ NaN \text{ or } inf & \text{if} \quad e = 0xFF \\ (-1)^s \times 2^{(e-127)} \times 1.m & otherwise \quad \text{(normal)} \end{cases} \tag{3}$$

Here, s, e, and m correspond to the sign, exponent, and mantissa of the FP32 value. As shown in Equation 3, the specification primarily follows two different modes, normal and subnormal, depending on the value of the exponent. Other floating point formats supported in DNN accelerators also follow a similar specification but with different numbers of exponent and mantissa bits, each making different trade-offs in the range and precision of values it can represent.

To analyze the sensitivity of model accuracy towards different floating point specifications, we define a data type policy in Arbitor of a generic floating point specification with configurable exponent and mantissa bits based on the IEEE 754 standard. We use *eXmY* to represent a floating point specification with 1 sign bit, X exponent bits, and Y mantissa bits. Using the custom floating point format, we train the DNN models in two ways. First, we train and measure the accuracy of the models while only using the custom floating point format for the whole model. Second, we use a combination of the custom floating point and FP32 arithmetic following the mixed-precision training strategy described in Section 3. Under this strategy, all arithmetic operations are performed using the custom floating point except the reduction operations in algebraic functions like matrix multiplication, convolution, and batch normalization which are performed using FP32 arithmetic. Figures 6a to 6d shows the validation accuracy measured on models trained using custom floating point format by varying the number of exponent and mantissa bits.

Observation 1: DNN models can maintain their accuracy with much lower precision than many standard low-precision floating point formats supported in current DNN accelerators (e.g., BF16 and TF32). Additionally, the precision can be reduced even further by using single-precision arithmetic for reduction operations.

From our results, CRNN, GNN, Transformer, and ResNet-18 can train to the FP32 accuracy with *e6m6*, *e6m3*, *e6m4*, and *e6m6* respectively. That means, reserving 6 bits for the exponent is sufficient to represent the floating point values generated during the training of these models which is lower than the standard floating point formats like FP32 (*e8m23*), BF16 (*e8m7*), and TF32 (*e8m10*), but higher than FP16 (*e5m10*). Reducing the exponent and mantissa bits beyond these configurations either causes a drop in accuracy or makes the models fail to converge altogether. We also observe that the accuracy of the models is less sensitive towards mantissa bits than exponent bits. This reasserts the fact that an optimal numerical format for DNN training workloads should allocate more bits for exponent and less for mantissa to be able to represent a wide range of values. Moreover, the mantissa bits can be further reduced for CRNN, GNN, and ResNet-18 to *e6m1*, *e6m2*, and *e6m2* respectively using FP32 arithmetic for reduction operations. This suggests that the numerical errors in low-precision training are primarily contributed by reduction operations. Therefore, using a combination of low and high-precision floating point formats in training can provide higher training performance with little to no loss in accuracy.

Next, we evaluate the importance of handling subnormal numbers in floating point formats. The subnormal mode in the floating point specification is introduced to gracefully handle underflow for the values that fall between zero and the smallest floating point number normal mode can represent. However, handling the subnormal mode in floating point implementation adds extra complexity to the hardware design. Therefore, certain implementations like BF16 in TPUs [31] do not support subnormal mode. In Figures 6a to 6d, we show the model accuracy measured using the custom floating point specification but with the subnormal numbers set to zero.

Observation 2: Subnormal mode has negligible impact on the model accuracy.

Comparing the accuracy measurements with and without subnormal mode, we observe that the subnormal numbers can often be safely ignored on configurations with sufficient exponent bits without affecting the model accuracy in almost all cases. This is due to the fact that floating point specifications with larger exponent bits have a smaller subnormal range and therefore approximating a small range of subnormal numbers to zero only introduces minimal numerical error in low-precision training. However, in certain cases like ResNet-18, low-precision training without subnormal mode can cause a slight drop in the accuracy of about 1.46%. We also observe that this accuracy loss can be compensated by adding more mantissa bits. For instance, ResNet-18 can achieve the baseline accuracy using *e6m8* without subnormal compared to *e6m6* with subnormal mode.

5.2.2 Posit as an Alternative for Floating Point

Even though floating point formats are the industry standard, there have been proposals for alternative numerical formats in the literature. Posit [25] is one such example and is a novel data type designed as a direct drop-in replacement for IEEE standard 754 floating point formats. Posit format is purportedly more hardware-friendly with lower power use and a smaller silicon footprint and can perform more operations per watt and per dollar than floating points under the same hardware budget. Moreover, the original paper [25] and subsequent studies [42] have shown that Posit can represent decimal numbers more precisely than floating point format on common arithmetic and linear algebra operations. Therefore, Posit is considered to be a viable alternative for floating point in deep learning applications. In this section, we evaluate the effectiveness of Posit in DNN training with Arbitor.

An n-bit Posit format with es exponent bits, abbreviated as $P(n,es)$, is represented using a sign (s), regime (k), exponent (e), and fraction (f) bits as follows:

$$value = (-1)^s \times 2^{2^{es} \times k} \times 2^e \times (1.f) \quad (4)$$

Similar to the floating point sensitivity study, we specify a data type emulation policy in Arbitor using the software-based Posit implementation available in the BFP library [40]. Then we train CRNN, GNN, and Transformer models using Posit configurations with different n and es values. Figure 7 shows the validation accuracy measured on these models.

Observation 3: Contrary to the observations made in prior works [59], low-precision training with Posit does not yield better accuracy compared to floating point.

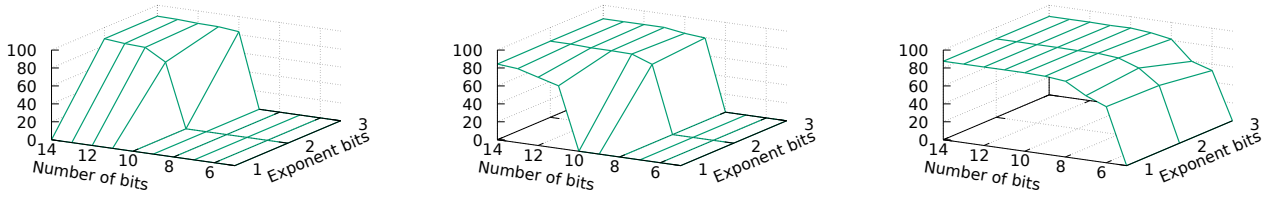
CRNN, GNN, and Transformer achieves the FP32 accuracy with *P(13,3)*, *P(11,3)*, and *P(9,2)* respectively. However, comparing a Posit and floating point configuration that uses the same number of bits in total, we observe they both converge to similar accuracy. This suggests that the higher precision of the Posit representation has limited benefits for DNN training workloads which are known to be tolerant of low-precision arithmetic. Despite this, we believe that Posit can still be a viable replacement for floating point in DNN accelerators due to its comparatively simpler and hardware-friendly design.

It is important to note that, the observation we make above goes against some of the prior works that claim that Posit can achieve better accuracy compared to floating point with a smaller hardware budget. For instance, Raposo et. al [59] have shown that 8-bit Posit can substitute 32-bit floating point for DNN training with no impact on accuracy. This underscores the importance of using a numerically accurate hardware emulation tool like Arbitor for such analysis. Moreover, the experimental evaluation conducted in this work was based on smaller models and datasets than what we use in our study. We believe our emulation tool can help hardware researchers to conduct accurate experimental analysis in the future and avoid making suboptimal design decisions.

5.3 Case Study #2: Approximate Computing

In this section, we use Arbitor to conduct a sensitivity study on a common approximate computing technique called *piece-wise linear approximation*. As described in Section 2, piece-wise linear approximation has been proposed as a cost-effective way to support non-linear algebraic functions like exponents and tanh that are common in natural language processing (NLP) models [35, 61, 69]. The key idea of this optimization is to break down curves of a non-linear function into pieces of line segments each approximating a part of the curve. Therefore, the fewer the pieces of line segments, the higher the numerical error in the approximation.

We analyze piece-wise linear approximation of the non-linear functions tanh and sigmoid in the CRNN model and their effect on the model accuracy by varying the number of pieces in the approximation. For this, we use the data type emulation policy in Arbitor to override the tanh and sigmoid function implementations of FP32 arithmetic with the piece-wise approximated version. Table 2a shows the validation accuracy curve of CRNN measured during the training.



(a) CRNN (b) GNN (c) Transformer

Figure 7: Validation accuracy of models trained using Posit format with different total bit widths and exponent bits

Observation 4: CRNN model can tolerate aggressive approximation on non-linear functions without affecting the accuracy.

We observe that even with an aggressive 3-piece approximation on both tanh and sigmoid functions, CRNN can maintain the baseline FP32 accuracy. This shows that piece-wise linear approximation is an effective optimization for DNN training. We also observe that a more aggressive 2-piece approximation on either function can significantly reduce the accuracy by up to 5%, and the accuracy drops further by 12% if both use 2-piece approximation. Since piece-wise linear approximation is often used in FPGAs and ASICs designed for NLP models, we believe our tool can be helpful in estimating the trade-offs of the approximation for specific models.

5.4 Case Study #3: Sparsity-Aware Processing

Finally, we use Arbitor to analyze how structured sparsity can affect the model accuracy. Sparsity-aware processing units are one of the recent innovations in DNN accelerators, e.g., Sparse Tensor Cores in Nvidia A100 [51]. Sparse tensor processing cores are designed to accelerate matrix multiplication over matrices with $N:M$ sparsity patterns, i.e., at most N values in every contiguous M block of values are non-zero. However, prior works [72] have pointed out that sparsity in DNN training is inherently unstructured, and enforcing any kind of structure to it in order to leverage Sparse Tensor Cores can affect the accuracy of the model.

We analyze the extent of these effects on ResNet-18 and Transformer with different sparsity patterns enforced on the weight parameter update operations during training. For this experiment, we define a masking policy in Arbitor with different N and M values. We use the absolute value of the weight parameter as the scoring function as it is one of the common heuristics for estimating the importance score in DNN pruning techniques [22]. Tables 2b and 2c shows the validation accuracy measured on ResNet-18 and Transformer.

Observation 5: The accuracy effects of sparse weight updates is highly model dependent. Moreover, enforcing more than 50% sparsity can have a significant impact on the model accuracy.

We observe that ResNet-18 can achieve close to the baseline FP32 accuracy with sparse weight updates. We also observe that ResNet-18 accuracy has slightly improved from the baseline accuracy with 50% sparsity patterns $1:2$, $2:4$, and $4:8$. We believe this improvement is due to the regularization effects of the sparse computation. On the other hand, Transformer shows a significant 4.3% drop in accuracy with 50% sparsity. This suggests that different models affect differently with sparsity-aware processing and require rigorous experimental analysis to accurately estimate the trade-offs. Applying patterns with more than 50% sparsity shows a significant drop in accuracy on both models by up to 7.5%.

6 Related Work

TensorQuant [45] is one of the first DNN emulation tools proposed and is specially designed for analyzing fixed point arithmetic primarily using layer-level quantization strategy. In addition, TensorQuant offers provision for fine-grained emulation strategy but requires users to write custom implementations for the operators in the DNN model. In contrast, Arbitor can support fine-grained emulation automatically with minimal effort from the user because of the compiler-based design that we follow. Moreover, due to the narrow focus on fixed point arithmetic emulation, the applicability of TensorQuant is limited to the analysis of DNN quantization methods. Arbitor, on the other hand, is capable of emulating arbitrary numerical formats using the data type emulation support and has a wider application. PositNN [59] is another DNN emulator built specifically for analyzing the efficacy of Posit numerical formats in DNN workloads. Therefore, like TensorQuant, PositNN also only has limited applicability. Moreover, PositNN uses a considerable amount of hand-written code for emulating Posit and only support a narrow range of DNN models. As we explain in Section 5.2.2, our analysis on a wider range of standard DNN models has refuted some

CRNN		
sigmoid	**tanh**	**Accuracy (%)**
3-piece	3-piece	88.12
2-piece	None	82.26
None	2-piece	84.92
2-piece	2-piece	70.95

(a) CRNN with piece-wise linear approximation

ResNet-18			
N \ **M**	**2**	**4**	**8**
1	94.07	93.64	92.99
2	-	94.09	93.59
4	-	-	94.19

(b) ResNet-18 with sparse weight updates

Transformer			
N \ **M**	**2**	**4**	**8**
1	82.11	78.78	75.45
2	-	82.94	80.72
4	-	-	82.94

(c) Transformer with sparse weight updates

Table 2: Validation accuracy of CRNN, ResNet-18, and Transformer with approximate and sparse computing

of the observations made using PositNN. Specifically, the authors of PositNN claim that 8-bit Posit format can substitute $32-bit$ floating point, which we find to be untrue on the models that we analyzed.

More recent emulators, QPyTorch [70] and GoldenEye [47] are both implemented on top of PyTorch. These tools are primarily designed for exploring low-precision formats on DNN training and provide emulation support for a wide range of numerical formats like floating point, fixed point, and block floating point [21] and follow layer-level emulation strategy with the help of the `hook` functionality in the framework. GoldenEye also provides hardware error injection support to evaluate the reliability of DNN accelerators. However, as described in Section 3, these tools are susceptible to numerical inaccuracies in emulation which can lead to incorrect assessment of the hardware design. We argue that numerically accurate emulation should be a necessary quality of hardware emulators. Moreover, unlike QPyTorch and GoldenEye, Arbitor is capable of supporting other common optimizations like approximate and sparse computing.

7 Discussion

We have presented a numerically accurate approach for emulating hardware optimizations and estimating their effects on DNN accuracy to guide the hardware design. Below, we discuss the generalizability and limitations of Arbitor, as well as our future plans to overcome these limitations.

Supporting a broader set of numerical data formats. Although we focus on case studies of floating point formats and Posit formats in this work, Arbitor is designed to support any arbitrary data formats through the generic data type emulation policy. Hence, other floating point formats like FP8 and fixed point formats like INT8 can also be emulated under this generic emulation policy. We have also extended Arbitor to support more complex block-based data formats, such as block floating point format [21] and MSFP [19]. These formats employ a block-based representation, where values within each block share the same exponent. Emulating such data formats is more challenging compared to regular data formats that only require considering individual data points, as the operations for each value depend on global information like the value of the shared exponent. The extensible design of Arbitor allows for adding compiler passes to rewrite operations for different blocks to support such data formats.

Supporting more front-end frameworks. We chose to implement Arbitor on top of XLA because we find it to be the most mature ML compiler with training support and is well-integrated with popular front-end frameworks like TensorFlow [3] and JAX [11]. However, the compiler-based approach is not tied to any specific compiler backend. We plan to migrate Arbitor to the latest OpenXLA that is based on the MLIR [38] infrastructure. Additionally, since the concept of operation-level emulation is not tied to any specific compiler backend, it is possible to apply the design of Arbitor to other ML compilers like TorchDynamo [67].

Supporting other complex hardware features. In this work, we demonstrate Arbitor's support for the three most common categories of hardware-level optimizations supported in DNN accelerators. However, it is worth noting that there exist intricate hardware features that are out of Arbitor's current scope. For instance, complex custom processing units, such as $3D$ cube in Huawei NPU [39], are currently not supported by Arbitor. We consider addressing these advanced hardware features as future work.

8 Conclusion

In this paper, we showcase that hardware researchers currently lack an accurate hardware analysis tool for empirically evaluating different design choices for DNN training accelerators. To fill this gap, we propose Arbitor, a hardware emulation tool for analysing common hardware optimization strategies like low-precision training, approximate computing, and sparsity-aware processing. Unlike prior emulators, Arbitor follows an extensible design and numerically accurate emulation support with the assistance of modern machine learning compilers like TensorFlow XLA. We subsequently demonstrate the utility of Arbitor by conducting an extensive sensitivity analysis on the aforementioned optimization strategies and their influence on the accuracy of popular DNN models.

Acknowledgments

We would like to thank our shepherd and the anonymous reviewers for their valuable feedback and comments. We also thank members of the EcoSystem lab for their support, inspiration, and constructive feedback during the development of this work. This project was supported by the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

Availability

The artifact of this paper is open-sourced on GitHub (https://github.com/arbitor-project/artifact).

References

[1] Forda: A dataset for time series classification.

[2] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[3] Martin 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, 2016.

[4] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *IEEE Access*, 7:78120–78145, 2019.

[5] Amazon Web Services. AWS Trainium.

[6] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin, 2015.

[7] Giorgos Armeniakos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. Hardware approximate techniques for deep neural network accelerators: A survey. *ACM Computing Surveys*, 55(4):1–36, nov 2022.

[8] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[10] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 129–146, 2020.

[11] 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.

[12] Jason Callaway. fra-txt-details. Kaggle Dataset.

[13] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. Deep positron: A deep neural network using the posit number system. In *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1421–1426, 2019.

[14] Cerebras. Cerebras Wafer-Scale Engine.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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, 2021.

[16] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. A survey of accelerator architectures for deep neural networks. *Engineering*, 6(3):264–274, 2020.

[17] 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, 2022.

[18] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications, 2014.

[19] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in neural information processing systems*, 33:10271–10281, 2020.

[20] David Kaplan. When hardware must just work, 2015.

[21] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 451–461, Red Hook, NY, USA, 2018. Curran Associates Inc.

[22] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks, 2019.

[23] Google. TensorFlow XLA, 2018.

[24] GraphCore. Introducing the Colossus MK2 GC200 IPU.

[25] Gustafson and Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J.*, 4(2):71–86, jun 2017.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[27] Meyr Heinrich, Lüthje Olaf, Holger Keding, and Coors Martin. Design and dsp implementation of fixed-point systems. *EURASIP Journal on Advances in Signal Processing*, 2002, 09 2002.

[28] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2016.

[29] James C. Hoe, Doug Burger, Joel Emer, Derek Chiou, Resit Sendag, and Joshua Yi. The future of architectural simulation. *IEEE Micro*, 30(3):8–18, 2010.

[30] Intel. Gaudi2: High Performance Training and Inference Solution.

[31] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[32] Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth

Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training. *CoRR*, abs/1905.12322, 2019.

[33] Keras. Keras Code Examples.

[34] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[35] Seok Young Kim, Chang Hyun Kim, and Seon Wook Kim. Applying piecewise linear approximation for dnn non-linear activation functions to bfloat16 macs. In *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4, 2021.

[36] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[37] Abhisek Kundu, Alex Heinecke, Dhiraj Kalamkar, Sudarshan Srinivasan, Eric C. Qin, Naveen K. Mellempudi, Dipankar Das, Kunal Banerjee, Bharat Kaul, and Pradeep Dubey. K-tanh: Efficient tanh for deep learning, 2019.

[38] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[39] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44, 2019.

[40] libcg. bfp - Beyond Floating Point.

[41] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2849–2858. JMLR.org, 2016.

[42] Peter Lindstrom, Scott Lloyd, and Jeffrey Hittinger. Universal coding of the reals: Alternatives to ieee floating point. In *Proceedings of the Conference for Next Generation Arithmetic*, CoNGA '18, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Zhenhong Liu, Amir Yazdanbakhsh, Taejoon Park, Hadi Esmaeilzadeh, and Nam Sung Kim. Simul: An algorithm-driven approximate multiplier design for machine learning. *IEEE Micro*, 38(4):50–59, 2018.

[44] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11966–11976, 2022.

[45] Dominik Marek Loroch, Norbert Wehn, Franz-Josef Pfreundt, and Janis Keuper. Tensorquant - a simulation toolbox for deep neural network quantization, 2017.

[46] Jinming Lu, Siyuan Lu, Zhisheng Wang, Chao Fang, Jun Lin, Zhongfeng Wang, and Li Du. Training deep neural networks using posit number system, 2019.

[47] Abdulrahman Mahmoud, Thierry Tambe, Tarek Aloui, David Brooks, and Gu-Yeon Wei. Goldeneye: A platform for evaluating emerging numerical data formats in dnn accelerators. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 206–214, 2022.

[48] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2017.

[49] MLPerf. MLPerf Training v0.6 Results, 2019.

[50] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[51] Nvidia. NVIDIA AMPERE GA102 GPU ARCHITECTURE.

[52] Nvidia. NVIDIA Tesla P100 White Paper.

[53] Nvidia. NVIDIA Turing GPU ARCHITECTURE.

[54] 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 Yang, Zach 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, 2019.

[55] Harry A. Pierson and Michael S. Gashler. Deep learning in robotics: A review of recent research, 2017.

[56] Jeff Pool and Chong Yu. Channel permutations for n:m sparsity. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 13316–13327. Curran Associates, Inc., 2021.

[57] PyTorch. RFC: Should matmuls use tf32 by default?

[58] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2018.

[59] Gonçalo Raposo, Pedro Tomás, and Nuno Roma. PositNN: Training Deep Neural Networks with Mixed Low-Precision Posit. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, page 7908–7912. IEEE, jun 2021.

[60] run.ai. FPGA for Deep Learning.

[61] Maicon A. Sartin and Alexandre C. R. da Silva. Approximation of hyperbolic tangent activation function using hybrid methods. In *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–6, 2013.

[62] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[63] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 197–209, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344, 2012.

[65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[66] Ruishuang Wang, Zhao Li, Jian Cao, Tong Chen, and Lei Wang. Convolutional recurrent neural networks for text classification. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6, 2019.

[67] Peng Wu. Pytorch 2.0: The journey to bringing compiler technologies to the core of pytorch (keynote). In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 1–1, 2023.

[68] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. Balanced sparsity for efficient dnn inference on gpu. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019.

[69] Babak Zamanlooy and Mitra Mirhassani. Efficient vlsi implementation of neural networks with hyperbolic tangent activation function. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(1):39–48, 2014.

[70] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. Qpytorch: A low-precision arithmetic simulation framework. *ArXiv*, abs/1910.04540, 2019.

[71] Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, and Yunji Chen. Fixed-point back-propagation training. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2327–2335, 2020.

[72] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n:m fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations*, 2021.

[73] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

[74] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network, 2019.

[75] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100, Sep. 2018.

A Artifact Appendix

Abstract

We provide the source code and scripts for Arbitor to reproduce the results of our experiments (in Section 3 and 4) and case studies (in Section 5) presented in the paper. While our paper explores three hardware optimizations: low-precision arithmetic, approximate computing, and sparsity-aware processing, the artifact focuses specifically on low-precision arithmetic. To this end, it contains the validation of accurate emulation of Arbitor. Additionally, the artifact conducts a sensitivity study between the validation accuracy and a subset of floating-point formats we study in the paper, similar to Figure 6. By following our provided instructions, users can expect to obtain results that closely align with those presented in the paper.

Scope

The artifact comprises two main components. Firstly, it generates the validation accuracy of training ResNet-18 using FP16 format emulated with Arbitor and QPyTorch and compares it to the native FP16 accuracy on GPU. This analysis highlights the negligible difference in accuracy between Arbitor-emulated FP16 and native FP16 while revealing a significant accuracy difference when using QPyTorch. Therefore, it could validate our claim that prior approaches like QPyTorch struggle to accurately replicate the numerical behavior of training with specific data formats, such as FP16, while Arbitor is designed to overcome this limitation.

Furthermore, the artifact automates the training process of GNN using a subset of floating-point formats evaluated in the paper, employing full low-precision training. This sensitivity study explores the variations in validation accuracy as the floating-point data format changes, thus validating the results presented in the corresponding section (Section 5) and supporting all associated claims. The results of all these training instances will be aggregated to generate a figure similar to Figure 6b but with fewer data points.

Contents

The artifact includes the source code of Arbitor, providing researchers with the ability to reproduce and modify the implementation. In addition to the evaluated ResNet-18 and GNN model, the artifact also includes the other two models, CRNN and Transformer, used in the case studies. Their corresponding datasets, including CIFAR-10 for ResNet-18, Cora for GNN, FordA for Transformer, and eng-fra for CRNN (as outlined in Table 1), are provided in the artifact. Therefore, researchers can readily evaluate these models and datasets, allowing for result replication, fast customization, and further investigation. Furthermore, the artifact provides a set of predefined data formats tailored to the experiments and case studies in the paper, including the floating-point and Posit formats, along with guidelines for incorporating these data formats in the training.

Hosting

The artifact can be downloaded from the main branch of GitHub link https://github.com/arbitor-project/artifact.

Requirements

A.0.1 Hardware requirement:

Arbitor requires the use of a multi-core CPU and an NVIDIA GPU with the Turing architecture or a more advanced counterpart to run the artifact. In our experiment, we employed four NVIDIA 2080Ti cards, but augmenting computation power by utilizing GPUs like NVIDIA A100, could further optimize the efficiency of artifact execution.

A.0.2 Software requirement:

The experiments provided in this artifact is prepared to run inside a docker container. We recommend using a machine with Ubuntu 20.04 with docker installed to reproduce the results.

Installation and Environment Setup

We provide docker files to set up the runtime environment for all the experiments.

- Install docker following the instructions in https://docs.docker.com/engine/install/ubuntu/.
- Make sure the machine has NVIDIA GPU(s) and the corresponding driver installed. If the NVIDIA driver is not installed, follow the instructions at NVIDIA tutorial to install it.
- Clone the git repository using the following command:
  `git clone --recursive https://github.com/arbitor-project/artifact`
- Build a docker image and enter the docker environment:
  `cd artifact && bash run.sh`

Experiment Workflow

We provide an end-to-end script to run everything all at once. When inside the docker environment, execute `e2e.sh` to run all the experiments and generate the results. The total execution may take several days to finish.

We also provide the step-by-step workflow as shown below:

1. Reproduce QPytorch ResNet-18 result with emulated FP16: `bash qpytorch.sh`
2. Reproduce Tensorflow ResNet-18 result with native FP16:
   `cd native_half && bash ./expr.sh resnet`
3. Reproduce Arbitor ResNet-18 result with emulated FP16:
   `cd /root/arbitor && bash ./expr.sh resnet`
4. Generate a subset of data formats for GNN training sensitivity study:
   `cd /root/ && bash ./gnn.sh`

Evaluation and Expected Results

Once the above execution has finished, two files, namely `results/validation.csv` and `results/sens.pdf` will be generated. `results/validation.csv` presents the final validation accuracy for three configurations: QPyTorch, native, and Arbitor FP16. It is expected that the validation accuracy of QPyTorch will be closely aligned with the FP32 baseline accuracy (93.78%), while both results of native and Arbitor are around 2% less than FP32 accuracy, consistent with the findings from Figure 3. Since the training process involves inherent randomness, it may be necessary to run multiple trials and compute an average for more accurate results. The file `results/sens.pdf` encompasses a figure depicting the relationship between accuracy, exponent bits, and mantissa bits. This is anticipated to bear resemblance to the first figure of Figure 6b, with fewer data points presented.

Experiment Customization

The emulated data format can be modified by modifying the `arbitor/data_format.sh` script to specify properties of data formats. Specifically, `F_OR_P` decides whether to emulate float or posit numbers. `ACC` represents the data type for accumulation, where `f32_acc` is to use FP32 to accumulate during a dot product, and `cus_acc` is to use the same type as computation for accumulation. In floating point configs, `EXP` and `MANTISSA` represent the bit-width of exponent and mantissa respectively. Setting `SUBNORMAL=_subnormal` causes subnormal numbers to be enabled during emulation and `SUBNORMAL=_wo_subnormal` otherwise. For Posit configs, `POSIT_NBITS` is the whole width of the number format, and `POSIT_ES` is the width of the exponent of Posit.

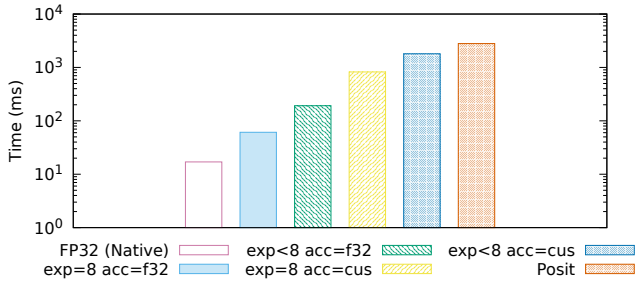


Figure 8: GNN training time with different number formats emulated by Arbitor

After changing `data_format.sh`, run
`bash ./expr.sh [gnn | transformer | crnn | resnet]`
to train the model with the specified data format.

B Overhead of Arbitor

In addition to accurate emulation, generating customized GPU kernels enables execution of Arbitor on GPU, in contrast to accurate software-based simulators that primarily rely on CPU-based execution. This allows Arbitor to leverage the high bandwidth and massive parallelism of GPU to achieve a level of performance that is more practical for executing DNN training workload compared to software-based simulators. To demonstrate the performance impact of Arbitor, we train the whole GNN model with different data formats emulated by Arbitor using the same experiment setup in Section 5.1. Figure 8 shows the training time for each step with different data formats, where `exp` is the width of exponent bits, and `acc=cus` or `acc=f32` represents whether the accumulation type is the same as computation type or is FP32. Emulating different data formats introduces different amounts of overhead, as the execution of emulated operations for each format requires a distinct number of cycles on the GPU. For the floating point format with 8 exponent bits with FP32 accumulation, Arbitor incurs a 3.59× overhead. When the data format is significantly different from FP32, such as Posit, Arbitor could introduce a slowdown of up to 164.7×. These results show that with the GPU acceleration, the performance of Arbitor is significantly better than software-based simulators and is practical for the accurate emulation of DNN training. Moreover, Arbitor can also leverage data parallel training to scale the training over multiple GPUs to compensate for the emulation overhead.

oBBR: Optimize Retransmissions of BBR Flows on the Internet

Pengqiang Bi
Shandong University

Mengbai Xiao *
Shandong University

Dongxiao Yu
Shandong University

Guanghui Zhang
Shandong University

Jian Tong
BaishanCloud

Jingchao Liu
BaishanCloud

Yijun Li
BaishanCloud

Abstract

BBR is a model-based congestion control algorithm that has been widely adopted on the Internet. Different from loss-based algorithms, BBR features high throughput since it characterizes the underlying link and sends data accordingly. However, BBR suffers from high retransmission rates in deployment, leading to extra bandwidth costs. In this work, we carefully analyze and validate the reasons for high retransmissions in BBR flows. In a shallow-buffered link, the packet losses are deeply correlated to both the bottleneck buffer size and the in-flight data cap. Additionally, bandwidth drops also cause unwanted retransmissions. Based on the analysis, we design and implement oBBR, which aims at optimizing the retransmissions in BBR flows. In oBBR, we adaptively scale the in-flight data cap and update the bandwidth estimate timely so that few excessive data are injected into the network, avoiding packet losses. Our Internet experiments show that oBBR achieves 1.54× higher goodput than BBRv2 and 39.48% fewer retransmissions than BBR-S, which are both BBR variants with improved transmission performance. When deploying BBR in Internet streaming sessions, oBBR obtains greater QoE than BBRv2 and BBR-S without incurring more retransmissions. To summarize, oBBR is designed to help a transmission session reach high goodput and low retransmissions simultaneously, while other CCAs only achieve one of them.

1 Introduction

Congestion control algorithms (CCAs) are essential for data transmission on the Internet. The loss-based CCAs like CUBIC [18] treat the packet loss as a signal of network congestion and thus throttle their sending rate. However, this significantly underutilizes the underlying link capacity since packet losses do not necessarily indicate congestion in the Internet nowadays. To effectively exploit the bandwidth resources, Google developed a model-based CCA based on measuring bottleneck bandwidth and round-trip propagation time, or BBR [8]. BBR ignores packet losses but adjusts its behaviors according to the estimated bandwidth and round-trip time (RTT). After switching to BBR from CUBIC, the throughput in Google's B4 network is consistently improved by 2-25× [9]. Since its release, BBR has been deployed on 22% of websites and accounts for over 40% of Internet traffic [36]. This attracts content providers, like YouTube [7] and Spotify [12], to adopt BBR.

While BBR exploits the bandwidth more effectively, this algorithm also leads to high retransmission rates. BBR is expected to send in-flight data at a volume of the bandwidth-delay product (BDP) only, i.e., operating at Kleinrock's optimal point [29], but it is still observed that BBR injects excessive data to the transmission channel because of bandwidth overestimation [22, 50]. To avoid the excessive data accumulated at the bottleneck, BBR imposes an upper bound to the volume of in-flight data at 2×BDP. But this results in a high packet loss ratio if the bottleneck buffer is not large enough to hold the excessive in-flight data. As a result, a large amount of data needs to be retransmitted, bringing extra bandwidth costs to content providers.

Directly reducing the upper bound of in-flight data in BBR is not feasible. By capping the in-flight data at 2×BDP, a BBR flow could "fairly" share the bandwidth with a loss-based flow (∼40%) when the bottleneck buffer is deep [14, 43, 52, 53]. If we limit the in-flight data in BBR further, it can no longer compete with the loss-based flows, thus achieving a degrading throughput.

Additionally, the high retransmission rates in BBR also result from bandwidth drop. BBR estimates the bottleneck bandwidth with the maximum delivery rate samples collected in an RTT-based time window. If the bandwidth drops, BBR will not update the bandwidth estimate until the outdated (and overestimated) samples leave the time window. While BBR keeps sending data and caps the in-flight data according to the overestimated bandwidth, the bottleneck buffer is quickly crammed and starts to discard packets. Moreover, the congestion at the bottleneck buffer enlarges the latest RTT samples

*Corresponding author

and thus the window size, making the old bandwidth samples expire even later.

In this paper, we propose oBBR, optimizing the retransmission rate and throughput in a BBR session. By extending the analysis model proposed in [52], we notice that the behaviors of a BBR flow are deeply correlated to the bottleneck buffer size and the in-flight data cap. Capping the in-flight data to a smaller value than 2×BDP mitigates the retransmissions in shallow buffers, but also weakens BBR's competitiveness in deep buffers. Furthermore, BBR can no longer compete with loss-based flows if the in-flight data is exactly 1×BDP. Following the analysis, oBBR detects if the bottleneck buffer is shallow once a packet loss event occurs. Then, oBBR reduces the in-flight cap accordingly so that the retransmissions are avoided. The in-flight cap is gradually recovered in case the packet loss is not caused by congestion. Furthermore, identifying a bandwidth drop event is not straightforward in BBR. Bandwidth samples with smaller values than the estimate are common because only the maximum is selected. In oBBR, we use consecutive samples of decreasing bandwidth or increasing latency to identify a bandwidth drop event. However, it is still possible that we falsely detect a bandwidth drop. We compare the delivery performance before and after the bandwidth estimate update. If the delivery performance degrades, we revert it to the old value.

We implement oBBR within a userspace implementation of Quick UDP Internet Connection (QUIC) [30], which is an appealing solution to network applications like video streaming due to its improved performance, high flexibility, and ease of deployment. We evaluate oBBR in both a lab environment and the Internet. In a stable network environment, oBBR reduces the packet loss ratio by up to 30× when compared to BBR. With 2% packet losses, oBBR achieves 14.65× higher goodput than BBRv2 which reacts to packet losses for alleviating retransmissions. In a network with fluctuating bandwidth between 40 Mbps and 10 Mbps, oBBR reduces the retransmissions by 52%. The realistic network emulation and the experiments on the Internet both show that oBBR achieves low retransmission ratios as loss-based algorithms, like BBRv2 and CUBIC, and high goodput as model-based ones, like BBR. Especially when continuously delivering data in a long Internet link, oBBR has the retransmission at ∼6.5% while gaining 1.54× more goodput compared to BBRv2, which retransmits 6.71% data. Compared to BBR-S, another BBR variant that also has reduced retransmissions and high goodput in our experiments, oBBR reduces retransmissions by 39.48%. We also implement oBBR in a video streaming system and measure the user's quality of experience (QoE) of streaming sessions on the Internet. Our experiments show that oBBR guarantees the QoE of a video session in terms of average quality, quality switches, and rebuffering ratio better than other CCAs including BBRv2 and BBR-S while suppressing the retransmission ratio as low as ∼4%. In summary, oBBR achieves both high goodput and low retransmissions in a transmission session,

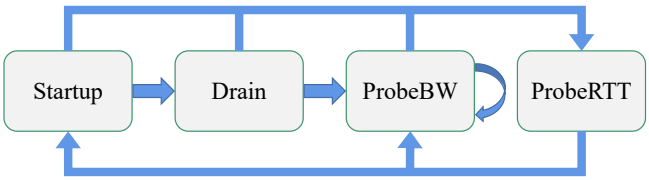


Figure 1: Overview of the BBR algorithm

while other CCAs only achieve one. The contributions of this paper are as follows:

- We carefully analyze why retransmissions are common in BBR flows. The high retransmissions in shallow-buffered links are deeply correlated to both the in-flight data cap and the buffer size. Moreover, the mechanism of BBR modeling the network becomes inaccurate if the bandwidth drops.
- We design and implement oBBR, optimizing the retransmissions of the BBR algorithm. oBBR adjusts the in-flight data cap according to the bottleneck buffer size and updates the bandwidth estimate promptly. Both reduce the excessive data sent by oBBR, avoiding packet losses.
- We carry out extensive experiments in both a lab environment and the Internet, justifying the design of oBBR. We also implement oBBR in a video streaming system on the Internet, and the results show that oBBR reaches higher QoE than BBRv2 while both have low retransmission ratios.

We briefly introduce how BBR works in Section 2 and analyze why the transmission rate of BBR flows is high in Section 3. The design of oBBR is presented in Section 4. In Section 5, extensive experiments are carried out to evaluate our design. Section 6 discusses related work and Section 7 concludes our work.

2 Background

BBR. BBR is a model-based congestion control algorithm released by Google in 2016 [8]. In a transmission session, BBR estimates the bottleneck bandwidth and the round-trip propagation time of the underlying link. To get unbiased estimates, BBR extracts the maximum from recent bandwidth samples and the minimum from recent RTT samples.

To regulate the traffic, BBR matches its average sending rate to the bandwidth estimate and caps the in-flight data by 2×BDP. Two key parameters, `pacing_gain` and `cwnd_gain`, are used in controlling the sending behaviors. `pacing_gain` inflates or deflates the sending rate while setting it to 1 means the sender transmits data at the rate of estimated bandwidth. This parameter is dynamically scaled in operation. `cwnd_gain` caps the in-flight data and is set to 2 in practice,

which means at most 2×BDP data are sent without acks. A BBR session always switches among four phases: *Startup*, *Drain*, *ProbeBW*, and *ProbeRTT*. An overview of the BBR algorithm is shown in Figure 1

1) Startup: BBR detects the available bandwidth by inflating its sending rate, where `pacing_gain` and `cwnd_gain` are both $2/ln2$. It switches to the *Drain* phase if the measured bandwidth stops growing.

2) Drain: BBR drains the excessive data queued at the bottleneck buffer because of the exponential growth of sending rate in the *Startup* phase. `pacing_gain` is set to $ln2/2$ until the in-flight data is less than 1×BDP. BBR then enters the *ProbeBW* phase.

3) ProbeBW: BBR spends most time in the *ProbeBW* phase, where `pacing_gain` is periodically set to {1.25, 0.75, 1, 1, 1, 1, 1, 1} and `cwnd_gain` is fixed to 2. Periodically inflating the sending rate helps BBR detect if the available bandwidth has increased, and deflating the rate could drain the queued data if the bandwidth is unchanged.

4) ProbeRTT: The *ProbeRTT* phase is independent of the other phases. BBR enters *ProbeRTT* once the RTT estimate has not been updated for 10 seconds. In this phase, BBR sends only 4 packets in-flight and observes their RTTs.

BBRv2. Though BBR gains high throughput in transmission, its mechanism has raised a few concerns: BBR does not react to the packet losses caused by network congestion and thus has high retransmission rates in links with a shallow bottleneck buffer; BBR does not share bandwidth with loss-based algorithms fairly; The throughput drops drastically in the *ProbeRTT* phase. To address these shortcomings, BBRv2 [11] is proposed and has been deployed in Google's internal network [10].

In BBRv2, packet loss events are considered signals of network congestion again. BBRv2 reduces the in-flight data cap multiplicatively if the packet loss ratio exceeds a threshold (2% in practice). But this also impairs the flow throughput as the loss-based CCAs do in a network environment with a high packet loss ratio (> 2%). In this work, we also compare oBBR with BBRv2, and the results are reported in Section 5.

3 Retransmissions in BBR

In this section, we analyze two major reasons that cause high retransmission rates of BBR flows in depth: 1) the underlying link is shallow-buffered, and 2) the available bandwidth drops in the transmission session.

3.1 Shallow-Buffered Link

Ideally, BBR operates at the optimal point of the transmission channel, i.e., no packets are queued in the bottleneck buffer and the latency approximates the physical delay. To achieve this, the in-flight data should equal 1×BDP of the channel. In practice, the bandwidth is usually overestimated [22, 50, 52], and thus the data would be gradually accumulated at the bottleneck buffer until packet losses. To avoid this, the sender employing BBR caps the in-flight data as

$$inflight = \texttt{cwnd\_gain} \cdot \widehat{RTprop} \cdot \widehat{BtlBw},$$

where $\widehat{RTprop}$ and $\widehat{BtlBw}$ are the estimates of round-trip propagation time and bottleneck bandwidth, respectively. `cwnd_gain` bounds the in-flight data to a small multiple of the BDP and is commonly set to 2. We can safely assume the volume of in-flight data reaches the upper bound most of the time because BBR overestimates its bandwidth share when competing with other flows [22].

However, $\widehat{RTprop}$ and $\widehat{BtlBw}$ are substantially affected by the competing flows, especially the ones with loss-based congestion control algorithms (CCAs). If the data from competing flows are also queued in the bottleneck buffer, $\widehat{RTprop}$ is then composed of the physical delay and the queue length, and $\widehat{BtlBw}$ is the link capacity proportional to BBR's buffer occupancy ratio [52]. The volume of in-flight data is calculated as

$$inflight = g \cdot (\frac{pq}{c} + l) \cdot (1-p)c,$$

where g represents `cwnd_gain`, q is the queue capacity of bottleneck buffer, p is the buffer ratio occupied by competing loss-based CCA flows, l is the RTT without congestion, and c is the link capacity. Then, the queued data of a BBR flow is

$$\begin{aligned} queued &= g \cdot (\frac{pq}{c} + l) \cdot (1-p)c - (1-p)cl \\ &= gq \cdot p(1-p) + (g-1) \cdot cl \cdot (1-p). \end{aligned} \tag{1}$$

By defining the occupied ratio of BBR flow at the bottleneck as $Q_r = \frac{queued}{q}$ and the relative size of bottleneck buffer to the BDP as $R = \frac{q}{cl}$, we have

$$Q_r(p; g, R) = gp(1-p) + (g-1)R^{-1}(1-p).$$

This helps us understand the behaviors of BBR with different configuration sets of g and R. We plot Q_R *vs.* p in Figure 2.

The left side of Figure 2 shows how a BBR flow behaves if $g = 2$, the current setup in BBR implementations. When the bottleneck buffer is as deep as 16×BDP, the BBR flow occupies almost half the buffer no matter how many loss-based flows coexist, which has been confirmed in prior studies [52]. If we reduce R, the shallower buffer will be more occupied by the BBR flow. When the buffer is as shallow as 1×BDP, the loss-based CCAs can not compete with BBR anymore. When the buffer size is lower than 1×BDP, we do not have enough space to hold the in-flight data and the packet loss ratio drastically increases.

If we set g to 1, expecting that no packet is queued when there is no competing flow, this also leads to another severe problem. As shown in the right of Figure 2, no matter how large the bottleneck buffer is, the BBR flow can not compete

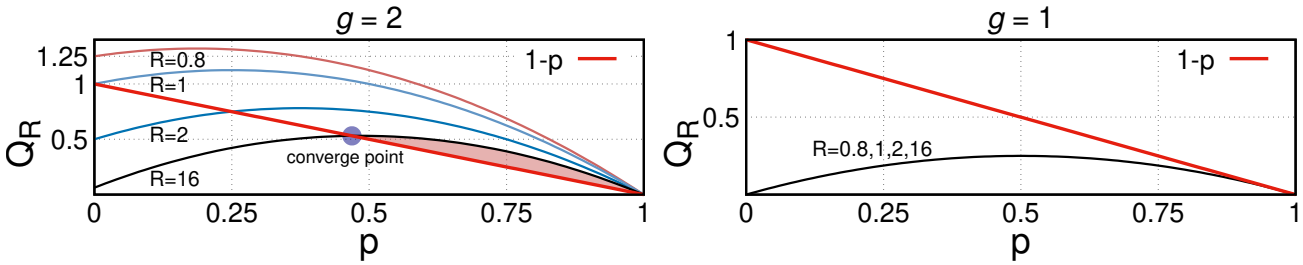


Figure 2: Q_R *vs.* p when setting g to 2 and 1. $1-p$ indicates the remaining buffer space for the BBR flow (from loss-based flows). We take $g = 2$ and $R = 16$ as an example. When $1-p$ is above the Q_R curve (at the left side of the figure), the buffer has enough space to hold the in-flight data of the BBR flow. As the loss-based CCA flows keep ramping up their in-flight data since no packets are discarded, p is increasing, which lets the BBR flow also accumulate more data in the buffer. The intersection of $1-p$ and the Q_R curve means that the buffer is fully occupied and loss happens. The loss-based CCA flows start to back off and then the BBR flow also backs off. As a result, the BBR flow operates around the converge point.

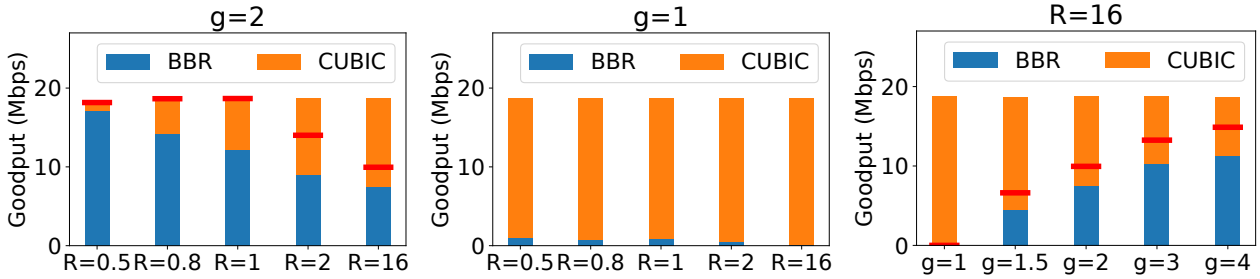


Figure 3: The average goodput of a BBR flow and a CUBIC flow with various configuration sets g and R

with the loss-based CCAs anymore, leading to poor performance.

Experimental verification: We set up the testbed in our lab environment based on nginx-quic [3], where we implement the BBR and CUBIC algorithms in the QUIC protocol. The details of the experimental setup are presented in Section 5.1. In the experiments, we simultaneously launch a BBR flow and a CUBIC flow. Both of them keep sending data without limitation from the application layer. The bandwidth is set to 20 Mbps and the RTT is set to 40 ms. By skipping the first 60 seconds, we measure the average goodput of two flows for 7 minutes when setting g and R to different values.

In Figure 3, the leftmost reports the average goodput of two flows when setting $g = 2$ and varying the bottleneck buffer size. We can see that the goodput proportion of the BBR flow decreases as R is greater, but it is still lower than the converge point $\frac{R-g+1}{gR}$ found in the model, which are marked as red lines on the bars. The reason is that BBR can not adjust its behavior as fast as the CUBIC does at the converging point: When two flows fully occupy the bottleneck buffer and packets are discarded, the CUBIC flow reduces its congestion window and the queue size shrinks, leading to that smaller RTTs are detected by the BBR flow. This immediately updates $\widehat{RTprop}$ and BBR calculates a smaller BDP. However, in the recovery phase, the CUBIC flow ramps up its congestion window quickly while BBR still uses the small BDP because the new RTT samples are greater than $\widehat{RTprop}$. The inaccurate RTT estimate expires in 10 seconds but before that, the CUBIC flow has occupied all free space in the buffer and backs off because of the packet loss again. Furthermore, BBR periodically enters *ProbeRTT* which reduces the in-flight data to 4 packets, also yielding bandwidth to the competing flows. But since the BBR ignores packet loss, this results in the BBR flow evicting the CUBIC flow and starting to experience apparent packet losses in a shallower buffer. When R is reduced to 0.5, BBR dominates the bandwidth and the packet loss ratio increases to 13.36%. The middle of Figure 3 confirms that when g is set to 1 and no matter how large the bottleneck buffer is, BBR can hardly compete with CUBIC and almost all the bandwidth is used by the CUBIC flow. The rightmost of Figure 3 shows the goodput of two flows when setting the bottleneck buffer to 16×BDP and varying g, where BBR gains more goodput with increasing g.

Based on the models and the experiments, we can derive that the packet losses of a BBR flow in a shallow-buffered link are correlated to g and R. If we reduce g, a BBR flow only evicts the coexisting loss-based flows and experiences packet losses in a shallower buffer. But this weakens the

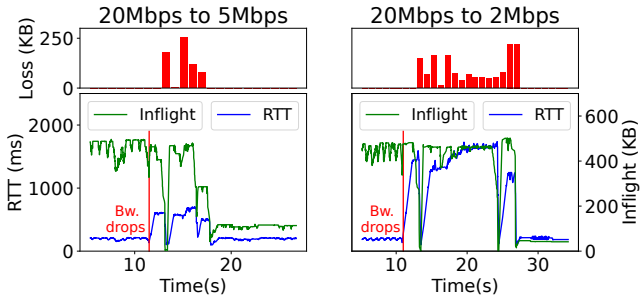


Figure 4: The network characteristics of a BBR flow when the bandwidth drops from 20 Mbps to 5 Mbps or 2 Mbps.

competitiveness of BBR in deep buffers. When g decreases to 1, BBR can hardly compete with loss-based flows no matter how large the bottleneck buffer size is. If set it to a high value, BBR flows have to sustain a high packet loss ratio when the bottleneck buffer is shallow.

3.2 Bandwidth Drop

BBR collects the delivery rate samples in a time window (typically 10 RTTs) and chooses the maximum as the bandwidth estimate. The rationale behind this is that acks can only be delayed. However, such a mechanism makes BBR react to the bandwidth drop sluggishly. When the bandwidth decreases, BBR keeps sending data at a high rate until the overestimated samples expire after 10 RTTs. This ramps up the queue size at the bottleneck buffer and leads to packet loss once the queue is crammed. More severely, as the packet delivery delay is extended due to the congestion at the bottleneck, the overestimated bandwidth samples expire in a longer period.

We conduct experiments to observe how a BBR flow reacts to the bandwidth drop in our testbed. In the experiments, the bandwidth is initially 20 Mbps, the RTT is 100 ms, and the bottleneck buffer is 200 KB. A BBR flow is launched to constantly send data. After ∼10 seconds since the flow starts, we throttle the bandwidth to a lower value (5 Mbps or 2 Mbps).

The experimental results are reported in Figure 4. The x-axis is the time elapsed since the flow starts in seconds and we plot the RTT samples in milliseconds, in-flight data volume in KB, and lost data volume in KB. We can observe that even the bandwidth drops, the in-flight data volume stays almost the same because of the unchanged $\widehat{BtlBw}$ and $\widehat{RTprop}$. While the actual bandwidth can not match the sending rate of BBR, the RTTs of packets rise sharply due to the increasing queue size at the bottleneck. The lost data volume also increases during this period. Only after 10 RTTs, i.e., ∼7 seconds on the left of Figure 4 (the RTT samples rise to ∼700 ms) and ∼15 seconds on the right of Figure 4 (the RTT samples rise to ∼1500 ms), the bandwidth estimate is corrected and the in-flight data match the BDP again. During the overestimation stage, the sudden drops of RTTs are caused by the *probeRTT* phase, but this does not help the BBR client realize the actual BDP.

4 Design

As discussed in Section 3, the main reasons for high retransmission rates in BBR are: 1) In a shallow buffer, the data in-flight could not be fully contained when g is fixed to a constant; 2) The bandwidth estimate is not timely updated when the bandwidth drops. To solve these problems, we propose to

- *Adaptive g*: We estimate the bottleneck buffer size with the RTT samples, and when a packet is discarded, the value of g is adjusted accordingly.
- *Timely bandwidth updates*: We identify the bandwidth drop based on both the RTT and delivery rate samples, updating the bandwidth estimate in time. This is reversible if the transmission throughput decreases.

4.1 Adaptive g

In practical BBR sessions, g is set to a fixed value of 2. This leads to the packet loss ratio drastically rising if the bottleneck buffer is shallower than $(g-1)\times$BDP. In this case, BBR packets are dropped because the in-flight data volume is larger than the buffer size q. By setting $p=0$ in Equation 1, the excessive data volume of a BBR flow is $(g-1)cl$. To avoid the packet loss events, we could adjust g to match the excessive data volume to the shallow buffer size q, i.e.,

$$g = \frac{q+cl}{cl}.$$

Whenever a packet loss event occurs, we can assume this results from a shallow bottleneck buffer. To scale g, we need to evaluate the buffer size by

$$q = c(l' - l),$$

where l' is the delivery latency of a packet that is queued at the end of the bottleneck buffer. Since the current buffer is full, l' can be reasonably estimated using the latest RTT sample RTT_{latest}. Then, we know that if

$$\frac{l'}{l} = \frac{RTT_{latest}}{\widehat{RTprop}} < g,$$

the bottleneck buffer is shallow.

Once we discover the packet loss is caused by a shallow buffer, we could scale g to

$$\min\left(1+\mu\cdot\frac{RTT_{latest}-\widehat{RTprop}}{\widehat{RTprop}}, g_{max}\right),$$

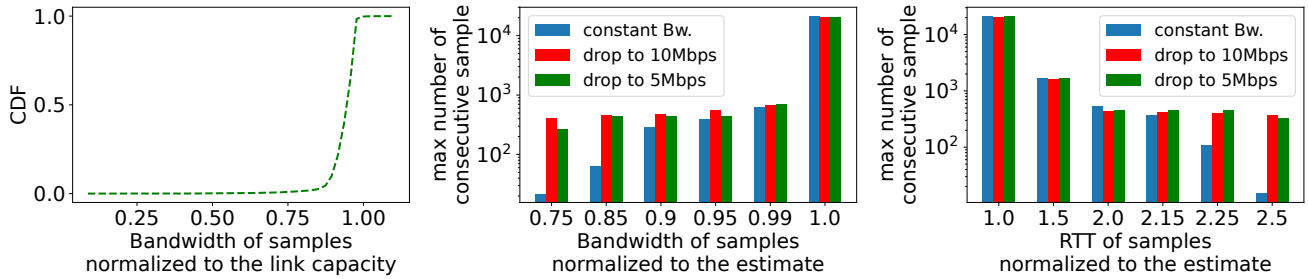


Figure 5: The network statistics of a BBR flow in various network environments. The network bandwidth is consistently set to 20 Mbps in the transmission session or decreases to 5/2 Mbps after 10 seconds. The leftmost figure presents the cumulative distribution function (CDF) of bandwidth samples normalized to the actual bandwidth when the bandwidth is constant. The figure in the middle shows the max number of consecutive bandwidth samples that are below the proportions of the estimated bandwidth. The rightmost figure shows the max number of consecutive RTT samples above thresholds of multiplicative RTTs.

where $0 < \mu < 1$ indicates the proportion of the shallow buffer attributed to the BBR flow and g_{max} is referred to as the fixed value used in deep buffer cases.

When the bottleneck buffer is non-shallow and there are competing flows, RTT_{latest} is expected to be greater than $g_{max} \times \widehat{RTprop}$ no matter if the packet loss event is caused by random loss or congestion. The $\min(\cdot)$ filter helps oBBR behave like the vanilla BBR in these cases if $g_{max} = 2$. If there are no competing flows, a smaller g but greater than 1 still guarantees the bandwidth is fully utilized. As a result, oBBR can adapt to the shallow buffer cases without affecting other scenarios.

Only scaling down g is not enough: 1) RTT_{latest} might be less than l'. For example, a packet loss happens before the shallow buffer is crammed. 2) The bottleneck might migrate to a deep buffer during the transmission session. Thus, we design a recovery stage to increase g back to g_{max}. Whenever an ack is received, we have

$$g = \min\left(g + \alpha \cdot \frac{S_{ack}}{BDP}, g_{max}\right),$$

where S_{ack} is the acked data size and α is a parameter preventing the congestion window growing too fast. In this way, after the congestion window is shrunk, it keeps increasing the value back until the packet loss happens again.

4.2 Timely Bandwidth Updates

BBR selects the maximum of bandwidth samples during the latest 10 RTTs as $\widehat{BtlBW}$. In the *ProbeBW* stage, BBR periodically inflates the sending rate to $1.25 \times \widehat{BtlBW}$ to verify if the actual bandwidth has increased. As long as a sample of higher bandwidth is observed, $\widehat{BtlBW}$ is updated so that BBR reacts to bandwidth increase timely. However, BBR fails to adapt to the bandwidth drop in time. Though the latest acks could show that fewer data has been received per unit time, these signals are ignored in estimating $\widehat{BtlBW}$. As a result, BBR keeps sending the data at a high rate until all samples of high bandwidth expire.

However, observing a sample of low bandwidth does not necessarily indicate the bandwidth has decreased. Acks might be delayed in the transmission and this is the reason of selecting maximum as the estimate. We collect bandwidth samples by running a BBR flow in the lab with 20 Mbps bandwidth and 100 ms RTT. The bandwidth samples are normalized to the realistic bandwidth and the results are shown in the leftmost of Figure 5. We can find that even for an experiment in the lab, 59% of samples are lower than $0.95 \times BtlBw$, and 24% of samples are lower than $0.85 \times BtlBw$.

Though the bandwidth samples could be inaccurate (we observe a sample of $0.09 \times BtlBw$ in the experiment), the samples that are highly deviated from the realistic bandwidth rarely appear consecutively. The middle of Figure 5 shows the max length of consecutive samples that are below varying thresholds proportional to the bandwidth estimate. We can see that for the constant bandwidth, at most 22 consecutive samples are observed below $0.75 \times \widehat{BtlBw}$. We also manually throttle the bandwidth to 10/5 Mbps in the transmission, and in both cases, the length of consecutive bandwidth samples could clearly signal the bandwidth drop. To effectively detect the decrease of bandwidth, we check if **there are k consecutive bandwidth samples less than $0.75 \times \widehat{BtlBW}$**. With a longer k, we are more confident to believe that the bandwidth has decreased rather than acks are occasionally delayed.

On the other hand, the bandwidth samples could be overestimated in realistic network environments. For example, routers could aggregate acks or have the capacity of handling burst traffic, both leading to calculating a greater bandwidth at the BBR sender. To detect the bandwidth drop even if the bandwidth samples are not reliable, we also use RTT samples. Since at most $g_{max} \times$BDP data are sent in-flight, the RTT samples should be about $g_{max} \times \widehat{RTprop}$ and samples with obviously greater values indicate the bandwidth estimate is not accurate anymore. We plot the max number of consecu-

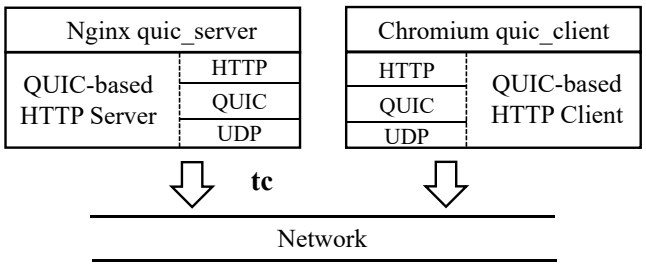


Figure 6: Overview of the testbed

tive RTT samples that are above thresholds of multiplicative $\widehat{RTprop}$ in the rightmost of Figure 5. When the bandwidth is constant, at most 15 samples are observed consecutively higher than $2.5\times\widehat{RTprop}$. This number increases to 373 or 328 if the bandwidth drops to 10/5 Mbps during the transmission, respectively. Thus, we also detect if **there are k consecutive RTT samples greater than** $(g_{max}+0.5)\times\widehat{RTprop}$. To the end, if **any of the two cases happen**, we think the bandwidth has decreased, and we use the average of k most recent bandwidth samples to update $\widehat{BtlBW}$.

Though we update the bandwidth estimate cautiously, it is still possible that an inaccurate value is used. So we monitor the delivery performance before and after the bandwidth update, reverting to the old value if the delivery performance decreases. Specifically, we calculate a delivery score of a fixed time interval T following

$$U = delivered - 10\times unacked,$$

where *delivered* is the amount of data acked and *unacked* is that not acked in T. We calculate the average score for $2T$ before and $2T$ after the bandwidth update. $\widehat{BtlBW}$ is reverted if the score drops.

4.3 Competitiveness Analysis

As discussed in Section 3.1, the competitiveness of BBR is determined by g and R. oBBR scales g to a smaller value once discovering the current link is shallow-buffered, while using a fixed g_{max} in deep buffers. By setting g_{max} to 2, oBBR behaves just like the vanilla BBR in deep-buffered links. A smaller or greater g_{max} weakens or strengthens its competitiveness.

In shallow-buffered links, the competitiveness of an oBBR flow is determined by μ, i.e., the excessive in-flight data proportional to the shallow buffer size. With a greater μ, oBBR is more competitive, but also has a higher risk of packet losses. Since the vanilla BBR selects a fixed g of 2, oBBR can hardly compete with it. But we can still expect that oBBR is less aggressive than BBR when competing with loss-based algorithms.

5 Evaluation

In this section, we evaluate oBBR and other peer methods in both a lab environment and the realistic Internet. We also build a video streaming system with oBBR that verifies our design also benefits network applications.

5.1 Experimental Setup

Implementation: We implement oBBR[1] in *nginx-quic* [3], which only has an RFC-defined congestion control algorithm.[2] In all experiments, we set α that controls the speed of recovery from the packet loss to 0.01. For detecting the bandwidth drop, we set k to 30 and T to 200 ms. We vary μ in 0.5, 0.75, and 1 to have oBBRs with different competitiveness, which is referred to as oBBR-0.5, oBBR-0.75, and oBBR-1 in the following discussion. In oBBR, we also set a lower bound 1.25 to `cwnd_gain` when the *ProbeBW* phase is probing for more bandwidth.

Additionally, we implement a couple of CCAs for comparison: **CUBIC** [18] is the default CCA in the Linux kernel, which considers packet loss as congestion and uses a cubic function to grow the congestion window. The vanilla version of **BBR** [8] manipulates its sending behaviors only depending on the estimates of bottleneck bandwidth and propagation round-trip time. **BBRv2** [11] is the successor of BBR proposed by Google, adding reactions to packet loss and greatly reducing retransmission rates. **BBR-S** [50] handles the bandwidth overestimation caused by the burst capacity of routers. It estimates the bandwidth at the 85th percentile of collected samples. **B3R** [44] is a BBR variant, which adjusts `pacing_gain` to regulate the sending rate in the *ProbeBW* phase. It aims at reducing the packet loss ratio when the bottleneck buffer is shallow. To compare all of these CCAs with oBBR on an equal footing, we implement these CCAs within nginx-quic. Implementing all of the CCAs within the same system architecture allows us to eliminate irrelevant factors that would otherwise complicate our comparison of CCAs.

It is worth noting that the *pacing* mechanism is not limited to BBR and its variants. The pacing mechanism sends data in a controlled manner so that the sending behavior becomes smoother. It has been proven effective in TCP connections with shallow bottleneck buffers [4]. So we also implement the pacing mechanism in CUBIC. Specifically, CUBIC calculates `pacing_rate` by `cwnd` / `srtt` × `pacing_ratio`, where `cwnd` is the congestion window and `srtt` captures the statistics of RTTs in both short-term and long-term. `pacing_ratio` is 2 in slow start and is 1.2 otherwise.

Testbed: We evaluate various CCAs in the client-server mode. nginx-quic is deployed as an HTTP server so that the transmission sessions in our experiments are HTTP sessions. In our lab environment, the server resides on a Linux machine

[1] Our prototype is available at https://github.com/bpq233/oBBR
[2] RFC 9002

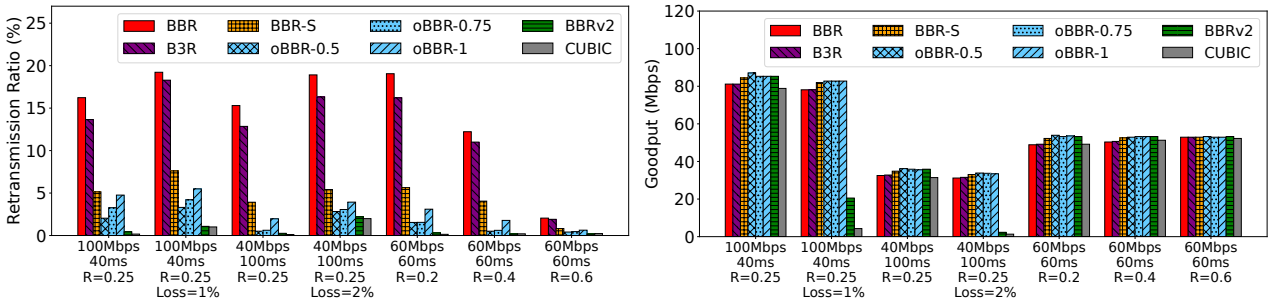


Figure 7: Performance of various CCAs in stable network environments

running Ubuntu 20.04 LTS with an Intel i5-7300HQ CPU @ 2.5 GHz (4 cores) and 16 GB RAM. The HTTP client is a simple implementation in the Chromium project [1] that also supports QUIC. For controlling the network conditions in our lab environment, *tc* [21] is used. In addition to changing the bandwidth, the latency, and the packet loss ratio, we adjust two parameters, `Limit` and `Burst`, to emulate the buffer size and the capacity of handling traffic exceeding the bandwidth at the bottleneck router. In experiments that use *tc*, `Burst` is set to 100 KB. An overview of the testbed is shown in Figure 6.

5.2 Retransmissions and Throughput

In the following experiments, we configure the client to fetch a data file of 1 GB via the HTTP protocol in a transmission session. The performance of various CCAs is measured in stable networks, variable networks, real network traces, and the Internet.

5.2.1 Stable Network Environment

Single flow: We first run various CCAs in stable network environments, where the bandwidth and the latency are set to {100 Mbps, 40 ms} or {40 Mbps, 100 ms}. We measure the retransmission ratio and the average goodput in a transmission session. The retransmission ratio is the volume of retransmitted data divided by the overall delivered data. R is 0.25 and the random packet loss ratio at 1% or 2% is also added. The results are shown in Figure 7. We can see that CUBIC and BBRv2 have the lowest retransmission ratios since they aggressively reduce the in-flight data when detecting packet losses. But both of them are vulnerable to random packet losses. Introducing a 1% random packet loss ratio reduces their goodput by 76% (BBRv2) and 94% (CUBIC), respectively. And the numbers increase to 94% (BBRv2) and 96% (CUBIC) with a 2% random packet loss ratio. On the other hand, BBR maintains its goodput across all cases but its retransmission ratios are also the highest, reaching 17.42% on average. This is because BBR does not throttle its sending rate until the packet loss ratio, whether caused by congestion or random losses, reaches a relatively high threshold of ~20%, which is not the case in our experiments. Across all scenarios, oBBR schemes consistently have high goodput, which is at least 82.76% of the link capacity. oBBR also suppresses its retransmission ratio of a session to a low value. For example, in the network with 40 Mbps bandwidth and 100 ms latency, oBBR keeps its retransmission ratios at 0.51% (μ=0.5), 0.62% (μ=0.75), and 1.97% (μ=1), respectively. By introducing 2% random packet losses, oBBR has the retransmission ratios at 2.82% (μ=0.5), 3.06% (μ=0.75), and 3.93% (μ=1). B3R lowers its sending rate to avoid excessive in-flight data, but this does not work in our experiments because of the bandwidth estimation. As a result, the retransmission ratios of B3R are similar to those of BBR. BBR-S more accurately estimates the bandwidth in our scenarios where the bottleneck router could handle burst traffic. Benefiting from this, BBR-S retransmits fewer data than BBR and B3R, but is still worse than oBBR schemes.

Then, we vary R to 0.2, 0.4, and 0.6, and fix the bandwidth and latency to {60 Mbps, 60 ms}. The results are also reported in Figure 7. With increasing bottleneck buffer size, BBR, B3R, and BBR-S reach a lower retransmission ratio accordingly because more in-flight data could be held in the buffer. The fewer retransmissions also improve their goodput. oBBR schemes have relatively high retransmission ratios when R=0.2, which are 1.54% (μ=0.5), 1.55% (μ=0.75), 3.10% (μ=1). This is due to we set a lower bound of 1.25 to g to match the 1.25×`pacing_gain` when probing for more bandwidth. When the R increases, the retransmission ratios of oBBR decrease again. For instance, when R=0.4, the retransmission ratios of oBBR are 0.51% (μ=0.5), 0.61% (μ=0.75), 1.78% (μ=1). BBRv2 and CUBIC still have the lowest transmission ratios while BBRv2 flows have goodput (similar to oBBR schemes) higher than CUBIC flows.

Multiple flows: We simultaneously launch at most 5 flows with the same CCA and measure the retransmission ratio and goodput per flow. In the experiments, the bandwidth and the latency are set to {100Mbps, 40ms}, and R is 0.5. No random packet loss is introduced. The results are shown in Figure 8. BBR still has the highest retransmission ratios, ranging from 8.41% (1-flow) to 15.25% (3-flows). As loss-based CCAs,

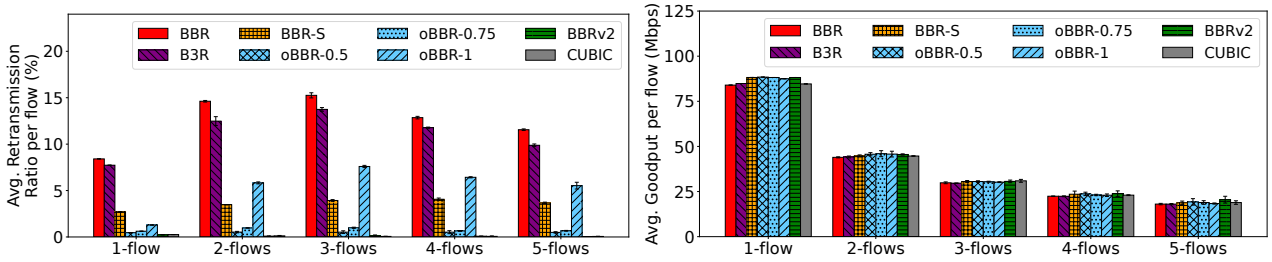


Figure 8: Performance of various CCAs in multi-flow scenarios

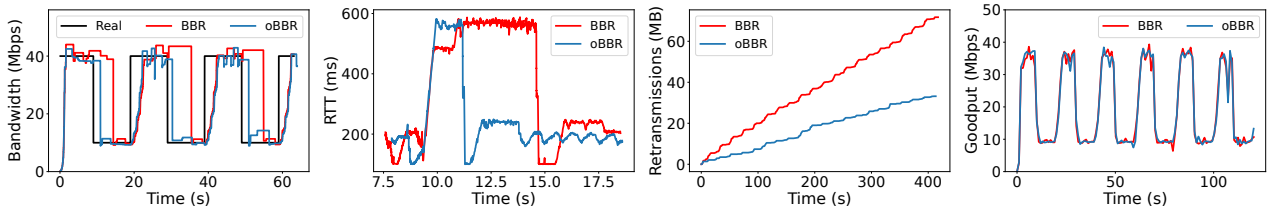


Figure 9: Performance of BBR and oBBR when bandwidth varies in a transmission session

CUBIC and BBRv2 retransmit few packets. When launching multiple BBR-like flows, i.e., BBR, B3R, BBR-S, or oBBR, higher retransmission ratios per flow are observed. The reason is that for each flow, the bandwidth overestimation is more severe: while some of the flows periodically switch to the drain cycle (`pacing_gain=0.75`) of *ProbeBW* or to *ProbeRTT* and yield bandwidth, the other flows obtain bandwidth estimate samples higher than the current ones. These overestimated samples are continuously used even though the co-existing flows scale back their sending rates. Such overestimations are alleviated when the number of flows increases (4-flows and 5-flows) because less bandwidth is allocated to each flow. For oBBR, if we set μ to 1.0 which intends to send excessive data exactly matching the bottleneck buffer, the overestimation easily results in the in-flight data being more than that. As a result, the retransmission ratio per flow of oBBR-1 grows to 7.59% at most (3-flows). When setting μ to 0.5 and 0.75, the bandwidth overestimation can hardly affect the transmission since oBBR leaves a margin in the bottleneck buffer on purpose. The highest retransmission ratios of oBBR-0.5 and oBBR-0.75 are 0.52% (2-flows) and 0.99% (3-flows), respectively. We also report goodput per flow in Figure 8, and it shows that all CCAs could effectively exploit and fairly share the link capacity.

5.2.2 Variable Bandwidth

We also set dynamic bandwidth to observe if oBBR reacts to the bandwidth drop timely as expected. In the experiments, the latency is 100 ms and the bottleneck buffer is 300 KB. We periodically change the bandwidth between 40 Mbps and 10 Mbps every 10 seconds. We evaluate oBBR and BBR, and the results are shown in Figure 9. The left two figures plot

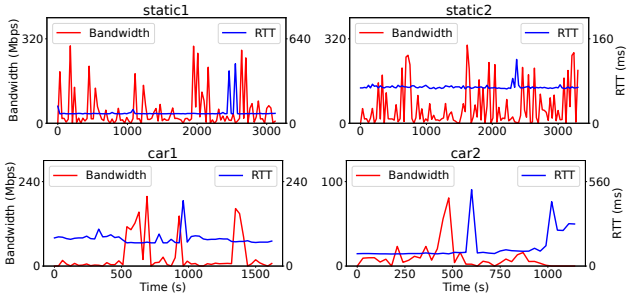


Figure 10: Characteristics of the realistic network traces

the bandwidth samples and the latency samples collected at the sender, respectively. We can see that oBBR reacts to the bandwidth drop more promptly as the bandwidth samples follow the real bandwidth change (the red line) in ~2 seconds. BBR only summarizes a more accurate bandwidth estimate after ~6 seconds because the high bandwidth estimate not only expires after 10 RTTs but also prolongs the latest RTT samples to ~600 milliseconds. While BBR sends in-flight data exceeding the bottleneck buffer, packet losses happen. Figure 9 also shows that BBR spends 2.09× more bandwidth than oBBR on retransmitting lost data while both of them achieve almost the same goodput in the transmission session.

5.2.3 Realistic Network Traces

We also test various CCAs in our lab by emulating the real network conditions following publicly available traces [41], which are collected from a major mobile operator in Ireland. Four traces are selected: *static1*, *static2*, *car1*, and *car2*. *static** means the trace is collected from a static object, probably a

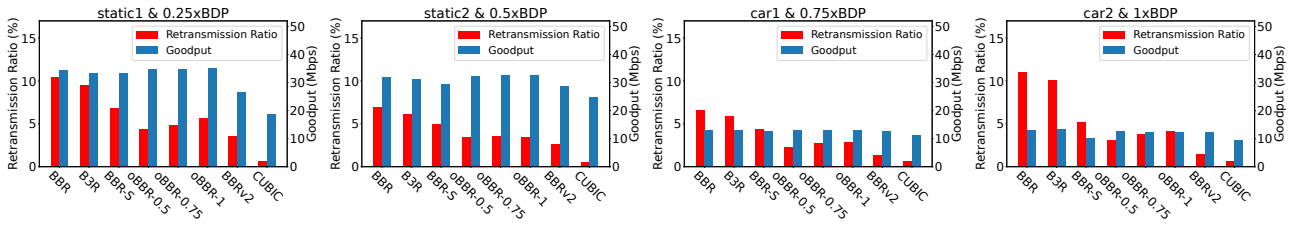


Figure 11: Performance of various CCAs in the testbed emulating the realistic network

Table 1: Characteristics of the realistic network traces

	avg. bandwidth	avg. RTT	duration
static1	45±62 Mbps	80±47 ms	3,137 s
static2	70±80 Mbps	69±6 ms	3,322 s
car1	32±64 Mbps	78±18 ms	1,645 s
car2	10±15 Mbps	118±78 ms	1,155 s

PC, and *car** means the trace is collected from a moving object, probably a car. Table 1 characterizes the traces and Figure 10 shows how the bandwidth and RTT vary over time.

We set R to different values for these traces. It is worth noting that no random packet losses are introduced in the emulations. We record retransmission ratios and average goodput of various CCAs. The results are reported in Figure 11. We can see that BBRv2 and CUBIC still have the lowest retransmission ratios in all cases, and more interestingly, they have lower goodput when compared to the oBBR schemes because oBBR reacts to the network dynamics more timely. Other performance trends are similar to those in stable network environments: BBR and B3R have the highest retransmission ratios. BBR-S has lower retransmission ratios but is still worse than oBBR. oBBR achieves the highest goodput in three out of four traces (35 Mbps of oBBR-1 in *static1*, 32 Mbps of oBBR-1 in *static2*, and 13 Mbps of oBBR-0.75 in *car1*).

5.2.4 Competitiveness

In this section, we measure how oBBR competes with other CCAs. We compare oBBR to BBR, CUBIC, and BBRv2, respectively. We also set μ to {0.5, 0.75, 1} for observing how this parameter affects oBBR's competitiveness. In experiments, the bandwidth is 40 Mbps and the latency is 100 ms. R is set to 0.5 or 1.0. For a transmission session, two flows, where one must be an oBBR flow, are launched simultaneously. We record the goodput of each flow and terminate both flows once the 1 GB data file has been fully delivered.

The results are shown in Figure 12. When competing with BBR, oBBR can hardly gain bandwidth when the buffer is as shallow as 0.5×BDP because BBR sends more in-flight data. This can be alleviated by increasing μ. Setting μ to 1 helps oBBR share 33% bandwidth. When the buffer size increases to 1×BDP, BBR performs less aggressively and even setting μ to 0.5 lets oBBR take 40.5% bandwidth. When competing with CUBIC, oBBR is aggressive in shallower buffers. But oBBR yields more bandwidth to CUBIC than BBR which takes almost all bandwidth in a 0.5×BDP buffer. In a buffer of 1×BDP, CUBIC takes up to 46.6% bandwidth share. When competing with BBRv2 in a 0.5×BDP buffer, the trend of bandwidth shares is similar to CUBIC because both of them react to packet losses. But since BBRv2 probes RTT more frequently, it takes more bandwidth in a 1×BDP buffer.

5.2.5 Internet

We further carry out the data delivery experiments in the Internet environment. We deploy the server in a data center in Virginia, USA, and the client in Shandong, China. The server is equipped with a 2-core CPU @ 2.5 GHz and 4 GB RAM, and its maximum egress bandwidth is 80 Mbps. The operating system is Ubuntu 20.04 LTS. The average link latency is about 270 ms. We experiment with data transmission in the morning, afternoon, and midnight. The results are shown in Figure 13. From the figure, CUBIC has the lowest retransmission ratio at 0.16% because it is the most sensitive algorithm to packet losses, and it also has the lowest goodput of 6.30 Mbps. BBRv2 has a retransmission ratio of 6.71% which is higher than the oBBR schemes because it fails to adjust its behavior timely in such a complex network environment, and its average goodput is 17.66 Mbps since the unavoidable packet losses in long Internet links. By setting μ to 0.75, oBBR achieves the highest goodput of 27.17 Mbps, which is 1.54× higher than BBRv2. Our oBBR schemes also have low retransmission ratios at 6.30% (μ=0.5), 6.58% (μ=0.75), and 6.85% (μ=1), which means 39.48%, 36.79%, and 36.11% fewer data are retransmitted compared to BBR-S, another BBR variant reaching a high goodput (26.94 Mbps) and a low retransmission ratio (10.41%) in the Internet experiments. These experiments indicate that oBBR could fully exploit the Internet link capacity while suppressing the retransmissions effectively.

5.3 Video Streaming

We also evaluate oBBR in video streaming, which is the dominant data delivery service on the Internet today.

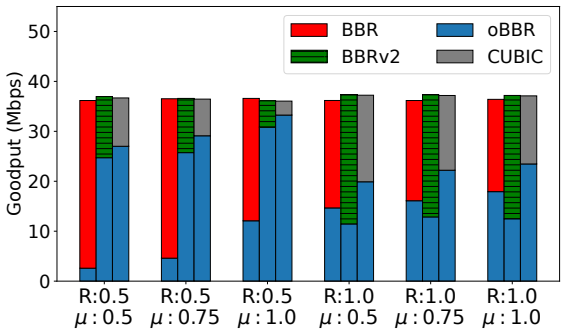


Figure 12: Competitiveness of oBBR against other CCAs

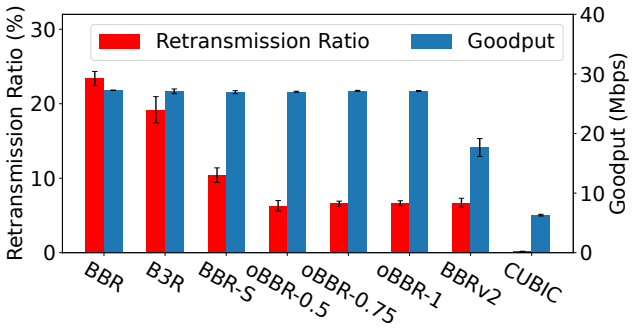


Figure 13: Performance of various CCAs on the Internet

5.3.1 Experimental setup

We build a video streaming system based on dynamic adaptive streaming over HTTP (DASH) [47], which uses HTTP as the vehicle for delivering video data. On the server, the test video is encoded into multiple bitrates and is further separated into segments of fixed length. Each of the segments is identified by a uniform resource locator (URL). We use *dash.js* [2] as the video player that fetches video segments to the client. In the player, the adaptive bitrate (ABR) algorithm dynamically switches between a buffer-based one and a throughput-based one to determine the quality of the next segment to download [46].

We use a 10-min video Big Buck Bunny as the test video, which is encoded into 10 bitrates of {254, 507, 759, 1013, 1254, 1883, 3134, 4952, 9914, 14931} Kbps. [3] For each bitrate level, the video is chunked into segments of 4 seconds. We put the server in Virginia and the client in Shandong, China. The experiments are carried out on the real-world Internet.

5.3.2 Metrics

Three metrics are used in the evaluation: average quality (AQ), quality switches (QS), and rebuffering ratio (RB).

Average quality: the average quality level (from 0 to 9) of all played segments. We expect the viewer to have a better quality of experience (QoE) with a greater AQ.

Quality switches: the number of segments with a lower quality than the previous one. A low QS means the visual quality does not fluctuate in playback, also indicating better QoE.

Rebuffering ratio: the proportion of time consumed in rebuffering. The viewers will stop watching if the rebuffering time is long. RB is calculated as $(\text{playback time} - \text{video length})/\text{video length}$.

5.3.3 Results and analysis

Figure 14 shows the performance of different CCAs in real-world video streaming sessions. For the retransmission ratio, BBR-S, BBRv2, CUBIC, and the oBBR schemes have similar performance, which is much lower than BBR and B3R. However, the low retransmission ratio of BBR-S, BBRv2, and CUBIC comes at the cost of low throughput, i.e., these algorithms fail to support the application layer to deliver high-quality videos. They have the lowest average quality, the highest quality switches, and the highest rebuffering ratio. For CUBIC, as it is too sensitive to packet loss, its throughput is suppressed at the lowest level, and the client almost selects the lowest quality level to fetch. This also decreases the rebuffering ratio of the CUBIC session. The oBBR also has low retransmission ratios (<4%) in the streaming sessions, but the user's QoE is not sacrificed. For example, by setting μ to 0.5, oBBR achieves the average quality of 6.37, the quality switches of 4, and the rebuffering ratio of 0.28%. Thus, the experiments show that oBBR benefits video streaming applications. It improves the QoE without incurring high retransmissions as BBR, which is friendly to content providers.

6 Related Work

BBR. BBR has been well-studied since it was proposed. It has been deployed in various networking scenarios, including mobiles [51], Wireless LANs [16], and clouds [17], and these research point out that the pacing mechanism in BBR may lead to poor performance. BBR is also implemented in MPTCP on Linux and the superior performance over other congestion control algorithms is observed [5]. The BBR-based MPTCP is further improved with respect to fairness and throughput in following studies [20, 35]. The performance of BBR against most existing CCAs has been extensively evaluated [31]. Such a wide range of research uncover that BBR still has problems.

Bandwidth overestimation in BBR. Since BBR selects the maximum of delivery rate samples within the latest 10 RTTs as the estimated bandwidth [56], it is easy to overestimate the bandwidth, which could result in network congestion. Chiariotti et al. [13] adopt the Adaptive Tobit Kalman Filter instead of the maximum filter for estimating bandwidth more accurately. Another study [19] suggests using Kalman Filter at

[3] Available at https://dash.akamaized.net/akamai/bbb_30fps/

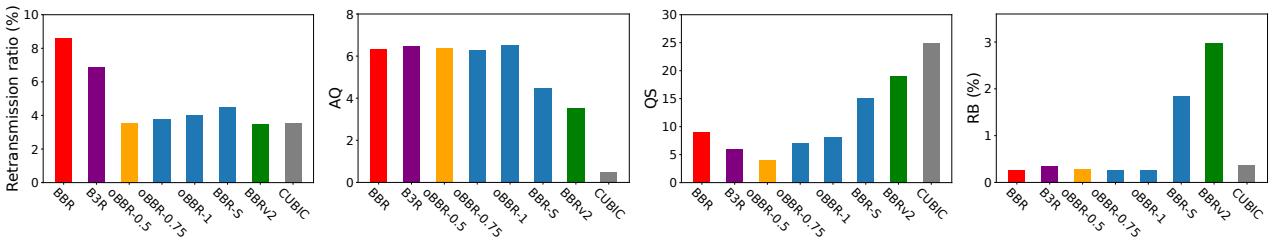


Figure 14: Performance of various CCAs in an Internet video streaming session

the receiver side and to communicate with the sender with the newly defined feedback frame of QUIC. A delayed bandwidth update strategy [48] also helps, which does not update the bandwidth estimate unless consecutive rate samples are received. BBRx [24] adjusts the estimated bandwidth based on the RTT deviation in an online learning manner. Different from these studies, we use a utility function to evaluate the delivery before and after the bandwidth update and revert to the old value if the throughput degrades.

High retransmissions in BBR. The high retransmission rates of BBR in shallow buffers have been recognized in prior studies [6, 14, 22]. To reduce the retransmissions, BBR-ACD [34] halves its congestion window when three consecutive identical RTT samples are received or the latest RTT sample is greater than 2×RTT estimate. BBR-A [33] is an extension to BBR-ACD that additionally decreases its pacing rate while the congestion window is reduced. In another study [45], the accurate propagation latency is detected when competing with CUBIC flows, and it cuts the congestion window to 1/3 of the in-flight data once there is packet loss. The existing optimizations for reducing high retransmissions in BBR flows apply loss-based multiplication to reduce the congestion window. oBBR sets the congestion window as BDP plus a proportion of bottleneck buffer so that the in-flight data will not keep decreasing even in a complex network, effectively guaranteeing the transmission throughput.

Unfairness of BBR. BBR has also been criticized for its unfairness when competing with loss-based CCAs [23, 42, 55]. Models [37, 52] are proposed for dissection. Researchers have found that a BBR flow occupies the same share of bandwidth regardless of how many loss-based flows coexist [52]. Furthermore, if too many BBR flows coexist, their advantage in throughput diminishes [37]. A learning-based model [27] has been proposed for determining the type of competing flows and mitigating the BBR's aggressiveness once the competitors are loss-based. Besides, a BBR flow gains a competitive advantage against another BBR flow if its propagation time is longer [32, 39, 40, 43, 49]. BBQ [32] enforces a cap to the span of the period that BBR pours more data for bandwidth probing. This bounds the advantage that a BBR flow with a long delivery latency can gain. BBR-E [28] reduces the in-flight data cap when the recent RTTs exceed a threshold. In another work [26], a factor γ is designed to compensate for the impacts of different RTTs, which makes the in-flight data from various flows almost the same. The fairness of BBR is out of the scope of this paper, but this is important and worth further exploring in future work.

BBRv2. To address these problems, Google introduced BBRv2 [11], which has been tested in a few studies [15, 25, 38, 54]. BBRv2 improves the fairness between flows with different RTTs and behaves less aggressively against loss-based CCAs [15, 38]. BBRv2 also reduces the high retransmission rates in BBR but at the cost of weakened resistance to packet losses [25]. In a network with bandwidth fluctuations, BBRv2 performs poorly [54]. In this work, we strive to solve the problems, i.e., degrading throughput and low responsiveness towards network dynamics, that still exist in BBRv2.

7 Conclusion

In this work, we carefully analyze the reasons for high retransmissions in BBR flows. We notice that the packet losses in shallow-buffered links are closely correlated to the in-flight data cap and the buffer size. Additionally, the slow reaction to bandwidth drops also makes BBR send excessive data to the transmission channel, thus leading to high retransmissions. To solve the problems, we design oBBR, which intelligently detects the bottleneck buffer size and scales the in-flight data cap accordingly, avoiding packet losses in the shallow-buffered links. oBBR also detects the bandwidth drop in an accurate and timely manner and tweaks its sending rate to avoid congestion. Extensive experiments in both a lab environment and the Internet have shown that oBBR significantly reduces retransmissions while still reaching a high data delivery rate.

Acknowledgments

We thank our shepherd, Eric Eide, and the anonymous reviewers for their constructive comments. This work has been partially supported by the grants from the National Natural Science Foundation of China (No.62102229 and No.62122042), the Natural Science Foundation of Shandong Province, China (No.ZR202206140010), and the Shandong Excellent Young Scientists Fund Program Overseas (No.2023HWYQ-045).

References

[1] Chromium. https://github.com/chromium/chromium/.

[2] Dash-Industry-Forum. https://github.com/Dash-Industry-Forum/dash.js/.

[3] Nginx-Quic. https://quic.nginx.org/.

[4] Amit Aggarwal, Stefan Savage, and Thomas E. Anderson. Understanding the Performance of TCP Pacing. In *Proceedings of the 19th IEEE Conference on Computer Communications (INFOCOM)*, pages 1157–1165, 2000.

[5] Phillipe Austria, Chol Hyun Park, Ju-Yeon Jo, Yoohwan Kim, Rahul Sundaresan, and Khanh D. Pham. BBR Congestion Control Analysis with Multipath TCP (MPTCP) and Asymmetrical Latency Subflow. In *Proceedings of the 12th Computing and Communication Workshop and Conference (CCWC)*, pages 1065–1069, 2022.

[6] Yi Cao, Arpit Jain, Kriti Sharma, Aruna Balasubramanian, and Anshul Gandhi. When to Use and When Not to Use BBR: An Empirical Analysis and Evaluation Study. In *Proceedings of the 19th ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 130–136, 2019.

[7] Neal Cardwell and Yuchung Cheng. TCP BBR Congestion Control Comes to GCP – Your Internet Just Got Faster. https://cloud.google.com/blog/products/networking/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster/, 2017.

[8] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control: Measuring Bottleneck Bandwidth and Round-Trip Propagation Time. *ACM Queue*, 14(5):20–53, 2016.

[9] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *Communications of the ACM*, 60(2):58–66, 2017.

[10] Neal Cardwell, Yuchung Cheng, Kevin Yang, Soheil Hassas Yeganeh, Priyaranjan Jha, Yousuk Seung, Luke Hsiao, Matt Mathis, Van Jacobson, Ian Swett, Bin Wu, and Victor Vasiliev. BBRv2 Update: Internet Drafts & Deployment Inside Google. https://datatracker.ietf.org/meeting/112/materials/slides-112-iccrg-bbrv2-update-00/, 2021.

[11] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, Ian Swett, and Van Jacobson. BBR Congestion Control. https://datatracker.ietf.org/doc/draft-cardwell-iccrg-bbr-congestion-control/02/, 2022.

[12] Erik Carlsson and Eirini Kakogianni. Smoother Streaming with BBR. https://engineering.atspotify.com/2018/08/smoother-streaming-with-bbr/, 2018.

[13] Federico Chiariotti, Andrea Zanella, Stepán Kucera, and Holger Claussen. BBR-S: A Low-Latency BBR Modification for Fast-Varying Connections. *IEEE Access*, 9:76364–76378, 2021.

[14] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 343–356, 2018.

[15] Jose Gomez, Elie Kfoury, Jorge Crichigno, Elias Bou-Harb, and Gautam Srivastava. A Performance Evaluation of TCP BBRv2 Alpha. In *Proceedings of the 43rd International Conference on Telecommunications and Signal Processing (TSP)*, pages 309–312, 2020.

[16] Carlo Augusto Grazia, Natale Patriciello, Martin Klapez, and Maurizio Casoni. BBR+: Improving TCP BBR Performance over WLAN. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–6, 2020.

[17] Phuong Ha, Minh Vu, Tuan-Anh Le, and Lisong Xu. TCP BBR in Cloud Networks: Challenges, Analysis, and Solutions. In *Proceedings of the 41th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 943–953, 2021.

[18] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[19] Habtegebreil Haile, Karl-Johan Grinnemo, Per Hurtig, and Anna Brunström. RBBR: A Receiver-Driven BBR in QUIC for Low-Latency in Cellular Networks. *IEEE Access*, 10:18707–18719, 2022.

[20] Jiangping Han, Kaiping Xue, Yitao Xing, Peilin Hong, and David S. L. Wei. Measurement and Redesign of BBR-Based MPTCP. In *Proceedings of the ACM SIGCOMM Conference Posters and Demos*, pages 75–77, 2019.

[21] Stephen Hemminger. Network emulation with NetEm. In *Proceedings of the Linux conf au*, volume 5, 2005.

[22] Mario Hock, Roland Bless, and Martina Zitterbart. Experimental Evaluation of BBR Congestion Control. In *Proceedings of the 25th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[23] Per Hurtig, Habtegebreil Haile, Karl-Johan Grinnemo, Anna Brunström, Eneko Atxutegi, Fidel Liberal, and Åke Arvidsson. Impact of TCP BBR on CUBIC Traffic: A Mixed Workload Evaluation. In *Proceedings of the 30th International Teletraffic Congress (ITC)*, pages 218–226, 2018.

[24] Jae Won Chung, Feng Li, and Beomjun Kim. BBRx: Extending BBR for Customized TCP Performance. In *Proceedings of the Proc. NetDev 0x12*, pages 262–276, 2018.

[25] Elie F. Kfoury, Jose Gomez, Jorge Crichigno, and Elias Bou-Harb. An Emulation-Based Evaluation of TCP BBRv2 Alpha for Wired Broadband. *Computer Communications*, 161:212–224, 2020.

[26] Geon-Hwan Kim and You Ze Cho. Delay-Aware BBR Congestion Control Algorithm for RTT Fairness Improvement. *IEEE Access*, 8:4099–4109, 2020.

[27] Geon-Hwan Kim, Yeong-Jun Song, and You-Ze Cho. Improvement of Inter-protocol Fairness for BBR Congestion Control Using Machine Learning. In *Proceedings of the International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 501–504, 2020.

[28] Geon-Hwan Kim, Yeong-Jun Song, Imtiaz Mahmud, and You-Ze Cho. Enhanced BBR Congestion Control Algorithm for Improving RTT Fairness. In *Proceedings of the 11th International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 358–360, 2019.

[29] Leonard Kleinrock. Power and Deterministic Rules of Thumb for Probabilistic Problems in Computer Communications. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 43.1.1–43.1.10, 1979.

[30] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan R. Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, pages 183–196, 2017.

[31] Jinting Lin, Lin Cui, Yuxiang Zhang, Fung Po Tso, and Quanlong Guan. Extensive Evaluation on the Performance and Behaviour of TCP Congestion Control Protocols under Varied Network Scenarios. *Computer Networks*, 163:106872, 2019.

[32] Shiyao Ma, Jingjie Jiang, Wei Wang, and Bo Li. Fairness of congestion-based congestion control: Experimental evaluation and analysis. *arXiv preprint arXiv:1706.09115*, 2017.

[33] Imtiaz Mahmud and You-Ze Cho. BBR Advanced (BBR-A) - Reduced Retransmissions with Improved Fairness. *ICT Express*, 6(4):343–347, 2020.

[34] Imtiaz Mahmud, Geon-Hwan Kim, Tabassum Lubna, and You-Ze Cho. BBR-ACD: BBR with Advanced Congestion Detection. *Electronics*, 9(1):136, 2020.

[35] Imtiaz Mahmud, Tabassum Lubna, Yeong-Jun Song, and You-Ze Cho. Coupled Multipath BBR (C-MPBBR): A Efficient Congestion Control Algorithm for Multipath TCP. *IEEE Access*, 8:165497–165511, 2020.

[36] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. The Great Internet TCP Congestion Control Census. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):45:1–45:24, 2019.

[37] Ayush Mishra, Wee Han Tiu, and Ben Leong. Are we heading towards a BBR-dominant internet? In *Proceedings of the 22th ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 538–550, 2022.

[38] Aarti Nandagiri, Mohit P. Tahiliani, Vishal Misra, and K. K. Ramakrishnan. BBRvl vs BBRv2: Examining Performance Differences through Experimental Evaluation. In *Proceedings of the 26th IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6, 2020.

[39] Wansu Pan, Xiaofeng Li, Haibo Tan, Jinlin Xu, and Xiru Li. Improvement of RTT Fairness Problem in BBR Congestion Control Algorithm by Gamma Correction. *Sensors*, 21(12):4128, 2021.

[40] Wansu Pan, Haibo Tan, Xiru Li, and Xiaofeng Li. Improved RTT Fairness of BBR Congestion Control Algorithm Based on Adaptive Congestion Window. *Electronics*, 10(5):615, 2021.

[41] Darijo Raca, Dylan Leahy, Cormac J. Sreenan, and Jason J. Quinlan. Beyond Throughput, the Next Generation: A 5G Dataset with Channel and Context Metrics. In *Proceedings of the 11th ACM SIGMM Conference on Multimedia Systems (MMSys)*, pages 303–308, 2020.

[42] Kanon Sasaki, Masato Hanai, Kouto Miyazawa, Aki Kobayashi, Naoki Oda, and Saneyasu Yamaguchi. TCP Fairness Among Modern TCP Congestion Control Algorithms Including TCP BBR. In *Proceedings of the 7th IEEE International Conference on Cloud Networking (CloudNet)*, pages 1–4, 2018.

[43] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. Towards a Deeper Understanding of TCP BBR Congestion Control. In *Proceedings of the 17th IFIP International Conference on Networking*, pages 109–117, 2018.

[44] Tarun Singhania, Wasim Arif, and Debarati Sen. B3R: A New Approach to BBR Congestion Control for Shallow Buffers. In *Proceedings of the Advanced Communication Technologies and Signal Processing (ACTS)*, pages 1–6, 2021.

[45] Yeong-Jun Song, Geon-Hwan Kim, and You-Ze Cho. Improvement of Cyclic Performance Variation between TCP BBR and CUBIC. In *Proceedings of the 25th Asia-Pacific Conference on Communications (APCC)*, pages 1–6, 2019.

[46] Kevin Spiteri, Ramesh K. Sitaraman, and Daniel Sparacio. From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 15(2s):67:1–67:29, 2019.

[47] Thomas Stockhammer. Dynamic Adaptive Streaming over HTTP: Standards and Design Principles. In *Proceedings of the 2nd ACM SIGMM Conference on Multimedia Systems (MMSys)*, pages 133–144, 2011.

[48] Bo Su, Xianliang Jiang, Guang Jin, and Haiming Chen. Rethinking the Rate Estimation of BBR Congestion Control. *Electronics Letters*, 56(5):239–241, 2020.

[49] Weifeng Sun, Minghan Jia, Guanghao Zhang, and Zun Wang. RFBBR: A RTT Faireness Awared Algorithm Based on BBR. In *Proceedings of the International Conferences on Smart Internet of Things (SmartIoT)*, pages 124–131, 2020.

[50] Santiago Vargas, Rebecca Drucker, Aiswarya Renganathan, Aruna Balasubramanian, and Anshul Gandhi. BBR Bufferbloat in DASH Video. In *Proceedings of the 30th The Web Conference (WWW)*, pages 329–341, 2021.

[51] Santiago Vargas, Gautham Gunapati, Anshul Gandhi, and Aruna Balasubramanian. Are Mobiles Ready for BBR? In *Proceedings of the 22th ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 551–559, 2022.

[52] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling BBR's Interactions with Loss-Based Congestion Control. In *Proceedings of the 19th ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 137–143, 2019.

[53] Ranysha Ware, Matthew K Mukerjee, Justine Sherry, and Srinivasan Seshan. The Battle for Bandwidth: Fairness and Heterogeneous Congestion Control. *USENIX Symposium on Networked Systems Design and Implementation Poster*, 2018.

[54] Furong Yang, Qinghua Wu, Zhenyu Li, Yanmei Liu, Giovanni Pau, and Gaogang Xie. BBRv2+: Towards Balancing Aggressiveness and Fairness with Delay-Based Bandwidth Probing. *Computer Networks*, 206:108789, 2022.

[55] Yuxiang Zhang, Lin Cui, and Fung Po Tso. Modest BBR: Enabling Better Fairness for BBR Congestion Control. In *Proceedings of the 23th International Symposium on Computers and Communications (ISCC)*, pages 646–651, 2018.

[56] Zhenzhe Zhong, Isabelle Hamchaoui, Rida Khatoun, and Ahmed Serhrouchni. Performance Evaluation of CQIC and TCP BBR in Mobile Network. In *Proceedings of the 21th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, pages 1–5, 2018.

Bridging the Gap between QoE and QoS in Congestion Control: A Large-scale Mobile Web Service Perspective

Jia Zhang[1,2,4], *Yixuan Zhang*[1,4], *Enhuan Dong*[1,3,4], *Yan Zhang*[1,4], *Shaorui Ren*[1,4], *Zili Meng*[1,4], *Mingwei Xu*[1,3,4], *Xiaotian Li*[5], *Zongzhi Hou*[5], *Zhicheng Yang*[5], *Xiaoming Fu*[6]
[1]*Tsinghua University,* [2]*Zhongguancun Laboratory,* [3]*Quan Cheng Laboratory,*
[4]*Beijing National Research Center for Information Science and Technology,*
[5]*Meituan Inc.,* [6]*University of Goettingen*

Abstract

To improve the user experience of mobile web services, various congestion control algorithms (CCAs) have been proposed, yet the performance of the application is still unsatisfactory. We argue that the suboptimal performance comes from the gap between what the application needs (i.e., Quality of Experience (QoE)) and what the current CCA is optimizing (i.e., Quality of Service (QoS)). However, optimizing QoE for CCAs is extremely challenging due to the convoluted relationship and mismatched timescale between QoE and QoS. To bridge the gap between QoE and QoS for CCAs, we propose Floo, a new QoE-oriented congestion control selection mechanism, as a shim layer between CCAs and applications to address the challenges above. Floo targets request completion time as QoE, and conveys the optimization goal of QoE to CCAs by always selecting the most appropriate CCA in the runtime. Floo further adopts reinforcement learning to capture the complexity in CCA selection and supports smooth CCA switching during transmission. We implement Floo in a popular mobile web service application online. Through extensive experiments in production environments and on various locally emulated network conditions, we demonstrate that Floo improves QoE by about 14.3% to 52.7%.

1 Introduction

Last decade has witnessed a dramatic increase in the use of mobile web services. The latest statistics demonstrate that more than 60% of global Internet users access web services with mobile devices [4]. Mobile web services are built on the transport layer, which typically employs a congestion control algorithm (CCA) that determines the data sending behavior and significantly affects the user experience. However, unsatisfactory performance of mobile web services has still been reported [40, 43, 49]. A recent measurement in 2020 shows that some web service users are still suffering from multi seconds request completion time [54], which also corroborates our observation in our web service in production (§5.3).

Our observation is that the key issue of the unsatisfactory performance in mobile web services is the mismatch between *what CCAs are optimizing* and *what the applications need*. Normally, CCAs optimize the Quality of Service (QoS) metrics describing the transport layer protocol delivery capabilities, i.e., delay, throughput, loss rate [22, 31], or a combination of these metrics [7, 19], in different network conditions. However, the applications do not really need an optimized QoS. Instead, they need a high quality of experience (QoE) for users. Specifically, for mobile web services, they have no idea what is *throughput* but only care about *request completion time (RCT)*[1]. The mismatch between QoE and QoS for current CCAs leads to their suboptimal performance.

However, optimizing QoE for CCAs is extremely challenging due to the following reasons. First, in terms of causality, the relationship between QoE and QoS is convoluted. For example, for small requests in web services, the increase of RTT will result in the degradation of the RCT. However, for large bulk requests, the effect of the increase of RTT on RCT is negligible. Therefore, without clearly understanding the relationship between QoE and QoS, directly optimizing the QoS may not be able to improve the QoE for the application. Second, in terms of timeliness, network conditions in QoS such as RTT and throughput are usually measured on a fine timescale (e.g., per packet). With this, CCAs can also make decisions on a short timescale. However, QoEs are usually perceived in a much longer timescale. RCTs in web services can only be calculated after the request completes, which usually takes seconds, during which, the CCA has already made numerous decisions. Thus, it is challenging for CCA to know which decisions are right and further correct its decisions (§2.1).

Our insight in this paper is not to directly optimize the CCA

[0]Jia and Yan are with Department of Computer Science and Technology, Tsinghua University. Yixuan, Enhuan and Zili are with Institute for Network Sciences and Cyberspace, Tsinghua University. Shaorui is with Department of Electronic Engineering, Tsinghua University. Mingwei is with Department of Computer Science and Technology and Institute for Network Sciences and Cyberspace, Tsinghua University. Xiaoming is with Institute of Computer Science, University of Goettingen.

[1]There are various QoE metrics of mobile web service, such as page load time. In this paper, we focus on the request completion time, i.e., the time interval between the request sent and the response fully received.

itself, but to introduce a shim layer between the application layer and transport layer to *select* the appropriate CCA for a better QoE. After decades of evolution of CCA, although there may not exist one CCA to fit all scenarios, we believe that there should always be at least one CCA that behaves well in a certain scenario. Selecting the CCA addresses the challenges above in two ways: First, instead of blindly optimizing those low-level instructions for CCAs with QoS, we could select the appropriate CCA based on the QoE. Second, we can perform the CCA selection at the same or longer timescale to fully and accurately utilize the information from QoE. Therefore, if we could always select and switch to the best CCA in the runtime, we will have the QoE directly optimized (§2.2).

However, it is challenging to propose a CCA selection mechanism for large-scale mobile web service due to the following reasons (§2.3).

- **Generating an optimal CCA selection policy is challenging.** The CCA selection policy, or in other words, mapping from the observed network conditions and application QoE metrics to the appropriate CCAs, is complicated. (1) The mobile network conditions can fluctuate, and are not easy to capture from the metrics observed on endpoints. (2) It is very difficult to model and characterize the CCAs, especially the recent proposed complicated CCAs [9, 19].
- **Switching between CCAs is nontrivial.** Different CCAs maintain different states. For example, BBR maintains the maximum delivery rate in the last 8 RTTs, but Cubic does not. If we need to switch between CCAs in the runtime, we should handle the states for a seamless switch carefully. Otherwise, if each CCA starts with the slow start, the QoE might be severely impaired during each switching.

To address the above challenges and provide better performance to the real applications, we propose Floo, a QoE-oriented mechanism for congestion control selection in large-scale mobile web services. Our key ideas are (1) to design *a QoE-oriented CCA selection mechanism*, and (2) to support *seamless CCA switching* during transmission. To turn our ideas into reality, we design several building blocks in Floo. First, we propose a reinforcement learning (RL)-based framework (to understand CCAs) that uses QoE as the selection criterion, and carefully selects both transport layer and application layer metrics (against network dynamic) to be jointly used in CCA selection (§3.2 and §3.3). Second, we devise a CCA switching mechanism to ensure the smoothness of switching by migrating the CCA phases and variables. The switching mechanism can be applied to traditional non-learning CCAs, and it is implemented with multiple classical CCAs in this paper (§3.4). Briefly speaking, Floo selects the optimal CCA for each connection according to QoE, and switches to a new, better CCA when the network condition changes (§3.1).

We implement Floo atop QUIC in the production environment of one Meituan's popular mobile web service application, Dianping, with O(10M) daily active users (§4.1). To make Floo work for real application scenarios, we collect real-world application traces for 14 days, including 35 million request logs. The traces are employed for analysis and training to reduce the gap between emulated environments and real world scenarios, enabling Floo to directly serve the real applications (§4.2). Extensive experiments demonstrate that Floo reduces the RCT by about 14.3% to 52.7% on average compared to using a static CCA. Further evaluation also shows that Floo is able to achieve satisfactory performance in the real world in different scenarios (§5).

In summary, our key contributions in this paper are:

- By demonstrating the difficulty of optimizing QoE for CCAs, we reveal the need for a practical QoE-oriented CCA selection mechanism (§2).
- We propose Floo, a QoE-oriented mechanism for CCA selection, which supports seamless switching on the fly for large-scale deployment of mobile web services (§3, §4).
- We deploy and evaluate Floo with Dianping service of Meituan. Our extensive experiments showed that Floo achieves consistent high performance under dynamic mobile networks (§5).

2 Motivation and Challenge

In this section, we use real-world mobile web service traces to demonstrate optimizing QoE for CCAs is extremely challenging in §2.1. Then, we present our design choices in §2.2 to address the mismatch between QoS and QoE. Finally, we elaborate on the challenges of designing the QoE-oriented CCA selection mechanism in §2.3.

2.1 Optimizing QoE for CCAs is extremely challenging

Convoluted relationship between QoS and QoE. QoE metrics are defined by applications. In contrast, QoS metrics focus on the descriptions of transport layer performance. The optimization of QoS is not consistent with that of QoE. For example, for large bulk requests, a reduction in RTT does not imply a QoE improvement [37]. Small request-intensive web pages are not that sensitive to throughput increase, since their total bandwidth need may still be small. As for the applications that apply recovery techniques such as FEC [32], packet loss also does not have a significant impact on QoE [25]. CCA has no idea what goal the application optimizes towards and whether application layer techniques are used. Therefore, the relationship between QoE and QoS is convoluted.

We conduct emulated experiments to demonstrate the convoluted relationship. Four well-known CCAs are considered: Cubic [23], BBR [12], Copa [9], and Westwood [14]. We use real-world traces extracted from Dianping to generate request-response messages (detailed in §4.2) on Mahimahi [41] emulated network paths. The WSP algorithm [44] is employed to compute the configurations of 100 different network path conditions (detailed in §4.3.1). Each CCA runs on each network path condition for 2 minutes. We calculate the metrics of Thpt, RTT, Power [27], etc., periodically at the sender according to transport layer acknowledgments. The results are presented

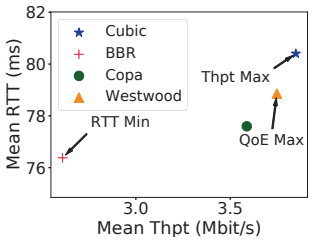


Figure 1: The mean throughput and RTT of the four CCAs achieved on a network path.

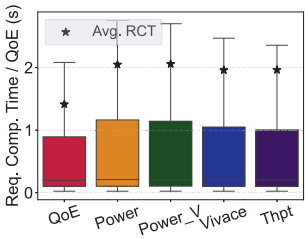


Figure 2: The RCT for all network path conditions following different CCA selection metrics.

Metrics	Definition
QoE	*Request completion time*
Power [27]	$Power = \frac{Throughput}{Delay}$
Power_V [28]	$Power\_V = \frac{Throughput*(1-LossRate)}{Delay}$
Vivace [19]	$Vivace = rate^t - b*rate*\frac{dRTT}{dT} - c*loss*rate$
Thpt	*Delivery rate*

Table 1: A QoE metric and four QoS metrics. QoE is calculated as the average RCT after the CCA convergence (20s after connection establishment). Power is the most common metric used for transport layer evaluation [7, 22, 31, 57]. Power_Variant [28] is one variant of Power, which also considers the packet loss rate. Vivace is used by an online learning CCA, and it is in form of utility function [19, 20].

in Fig. 1 and 2.

Fig. 1 shows the performance of the CCAs on a specific network path. Among the four CCAs, the CCA that achieves the highest throughput or the lowest RTT does not reach the best QoE (defined in Tab. 1). Then we consider the relationship between QoS and QoE on multiple network paths. For each network path condition, we select the best CCA following different metrics listed in Tab. 1. For each metric, we obtain all the RCT of the requests running the selected CCA for all network path conditions. Fig. 2 shows the Tukey boxplot for each metric. Results show that if the CCA is selected following the QoE metric, the RCT can be reduced by at least 27% on average, and none of the popular QoS metrics can achieve similar RCT to the QoE metric. This well demonstrate the convoluted relationship between QoS and QoE.

Mismatched timescale between QoE and QoS. The mismatch also exists in the time scale of the CCA and application optimization. CCAs collect fine-grained ACK information and make decisions at a granularity of packet-level or RTT-level. In contrast, QoE is measured and evaluated at a coarse granularity of the request level, usually around hundreds of milliseconds or even longer [9, 54]. As a result, it is difficult to map high time scale QoE to low time scale CCA behavior.

2.2 Design Choices

Instead of optimizing QoE for CCAs, we decide to design a shim, which selects an appropriate CCA aiming for better QoE. The QoE-oriented CCA selection approach addresses the mismatch between QoS and QoE:

- **Optimizing the real goal.** It is hard to make QoS-oriented CCAs optimize the QoE, because of the mismatch between QoE and QoS. However, with QoE metrics as the basis, the CCA selection shim allows the transport layer behavior to be optimized toward application layer objectives. As discussed in §2.1, the QoE is hard to be replaced by existing transport capability-oriented QoS metrics. Therefore, QoE is regarded as the real goal of our approach.
- **Time scale.** The CCA selection approach works above the transport layer to make decisions at a coarser granularity, understanding the QoE, and deciding which CCA to use. Specific sending rate/CWND increase or decrease decisions of the CCAs' are not necessarily closely coupled with QoE. Thus, the time scale mismatch is solved. The CCAs do not need to interact with the application layer and do not need to be modified.

2.3 Challenges

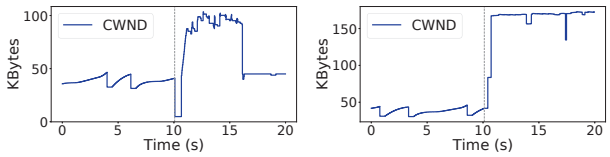
However, designing and implementing a CCA selection mechanism is non-trivial in a large-scale real-world deployment of mobile web service.

CCA Selection. Creating a mapping from the observed network conditions and QoE metrics to CCAs, or in other words, generating a CCA selection policy is challenging.

- **Fluctuating network conditions.** Under mobile networks, network conditions fluctuate due to wireless channel fading, user movement, or network congestion. Adapting to the dynamic network condition is challenging.
- **Empirical CCA characteristics.** The existing knowledge of the applicable scenarios of CCA is usually empirical [11, 15]. It is very difficult to model and characterize the CCAs, especially the recent proposed CCAs [9, 19]. Adapting to the complicated CCAs is challenging.

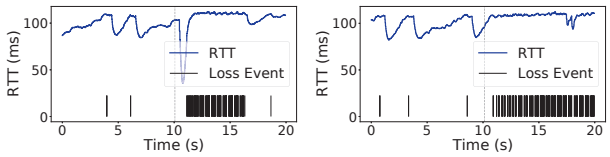
Smooth switching on the fly. Due to dynamic network conditions, CCA switching may occur during transmission. We consider two kinds of switching: Part Switching and Full Switching. While Full Switching makes the new CCA inherit all CCA-related variables, including connection-level variables (e.g., CWND/sending rate and RTT-related values), and CCA private state variables (e.g., fulled_pipe in BBR), Part Switching only inherits connection-level variables, and the private state variables of the new CCA are initialized from default values. To demonstrate their differences, we build a small testbed including two hosts and one switcher. The two hosts establish a QUIC connection, which continues to send massive data. One CCA switch event happens at the 10th second. As shown in Fig. 3, the switching without CCA state migration (Part Switching) has two problems:

- **Longer convergence time and performance deterioration.** Without CCA state migration, the new CCA starts at the slow start phase and the CWND or sending rate increases exponentially from the steady state of the previous CCA until it converges again. The path condition information required for the new CCA still needs time to be col-



(a) Cubic switches to BBR.

(b) Cubic switches to Copa.



(c) Cubic switches to BBR.

(d) Cubic switches to Copa.

Figure 3: Two demonstrations of Part Switching. The switch event happens at the 10th second. (a) and (b) show the CWND change over time. (c) and (d) show the smooth RTT changes over time and when loss events happen. Part Switching from Cubic to BBR leads to longer convergence time and performance deterioration. Part Switching from Cubic to Copa leads to distorted path estimation collected and results in abnormal behavior of the new CCA.

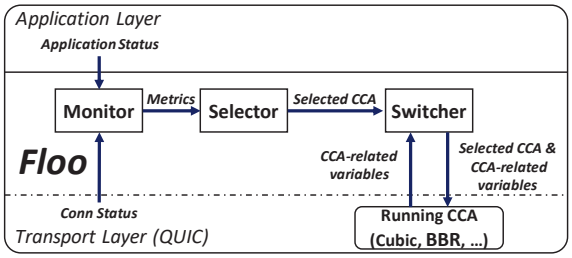


Figure 4: Floo Overview

Characteristic	Metric	Description
Btlbw	$Thpt\_max$	The maximum delivery rate
RTProp	RTT_min	The minimum RTT
Random Loss	$Loss\_rate$	|Lost Packet| / |Sent Packet|
Buffer	RTT_rate	Smooth RTT / Min RTT
RTT Variety	RTT_{var}	The variance of RTT samples
CCA	CCA_name	Current CCA
	$CWND/SendRate$	CWND or pacing rate
App-level	$Goodput$	The average Size / Duration of past responses
	$Unsent\_size$	Total bytes of the response waiting for writing to Send Buffer
	Qws	The average duration the response wait for writing to Send Buffer
	$Qws_{gradient}$	The gradient of the Qws samples
	$Bytes\_interval$	Bytes sent in the last SP

Table 2: The metrics collected by Monitor module.

lected and estimated. Worse still, since the previous CCA has converged basically, the new CCA's re-convergence will cause a lot of packets lost (about 14.8% within 2s after switching in Fig. 3c).

- **Distorted path estimation results in abnormal behavior of new CCAs.** The network condition observed by the new CCA reflects the condition after the previous CCA converged. It may not be consistent with the real path condition. For example, as shown in Fig. 3d, there is already a queue buildup in the buffer by Cubic when the switching occurs. Therefore, the *minRTT* observed by Copa is biased after the switching. In such a case, Copa is unable to make proper decisions to reduce the RTT or drain the queue. Fig. 3d shows that the high RTT may last for tens of seconds or even longer, which completely conflicts with Copa's design goal of low latency.

Therefore, smooth switching is necessary. Ideally, the new CCA should inherit all the CCA-related variables and continue to update them according to the newly observed network conditions after the switching. However, considering the more complex state design of emerging CCAs, and the personalized state variables, mapping the state of a certain CCA to a new one is much more challenging.

3 Design

3.1 Design Overview

Fig. 4 shows the overall architecture of Floo. We build the main building blocks of Floo atop QUIC, including Monitor module, Selector module and Switcher module. Monitor module collects information from both the transport layer and application layer. According to the state variables saving application statistics and connection statistics, Monitor module computes the metrics of interest in consecutive time intervals and feeds them to Selector module every selection period (SP) (§3.2). According to the metrics fed by the Monitor module, Selector module selects a CCA based on a pre-trained selection policy, which maps the metrics from Monitor module to appropriate CCAs. With the help of that policy, an appropriate CCA is selected to maximize the application QoE. The selection policy is pre-trained offline with RL methods (§3.3). After getting informed of the new optimal CCA, Switcher module conducts CCA state migration to complete a smooth switching from the current CCA to the new CCA (§3.4). Thus, an accurate and smooth switching of CCA is completed.

For a QoE-oriented CCA selection mechanism, one key design decision of Floo is how to incorporate QoE metrics into the mechanism. From emulated experiments in §2.1, we observe that it is difficult to replace QoE with QoS. As Fig. 2 shows, the CCAs selected with the QoE criterion achieve the lowest RCT, while the other four QoS metrics, Power, Power_V, Vivace and Thpt, have 31.01%, 31.34%, 27.87%, 28.01% higher RCT on average respectively. Therefore, our answer is to directly set QoE as the selection criterion.

3.2 Monitor

Monitor module gathers collectible statistics that can be measured and monitored, including the transport layer and application layer statistics. The transport layer statistics are collected from the information provided by ACK packets or the connection maintained state variables. As for the application layer statistics, we collect them from mobile web applications.

We consider the metrics shown in Tab. 2. The CCA selection mechanism is to select the optimal CCA for different network conditions. Therefore, we first consider the characteristics describing the network path conditions. They are

the former 5 rows of the table. We use bottleneck link bandwidth ($BtlBw$), round-trip propagation time ($RTProp$), RTT variety (RTT_{var}), packet loss rate ($RandomLoss$) and buffer size ($BufferSize$) to describe a network path. We estimate these objective network path conditions with the following collectible metrics: the maximum delivery rate ($Thpt_{max}$), minimum RTT (RTT_{min}), average loss rate ($Loss\_rate$), relative value of buffer size v.s. BDP (RTT_rate) and the RTT variety (RTT_{var}). The calculation of the metrics occurs in a slicing time window of 10s.

Note that the above metrics are only the observed metrics of the network paths and not the objective network path conditions. The relationship between the observed and objective values can be affected by the current running CCA. In order to accurately reflect the objective values of the network path, two additional types of information are collected. On the one hand, we record the current CCA, and $CWND$/$SendRate$ if applicable. On the other hand, we collect the information from the application layer, specifically the $Goodput$, $Unsent\_size$, Qws, $Qws_{gradient}$, $Bytes\_interval$ (defined in Tab. 2). It is worth noting that Qws is the response waiting time, which is the time between the response is generated on the server and the response data is written to the send buffer of the transport layer. Qws reflects the growth and drain of the queue in the send buffer. When the network condition worsens, the queue in the send buffer will pile up rapidly, making the rise in Qws. The calculation of these app-level statistics occurs every SP.

3.3 RL-based Selector

Selector module selects the optimal CCA based on the metrics passed by Monitor module. The selection policy, or the mapping from the Monitor module metrics to the optimal CCA, is pre-trained and saved in Selector module. We utilize an RL approach to build a prediction model as the selection policy, because RL and CCAs are similar, i.e., both of them continuously make decisions according to the changes of environment. In this section, we describe the RL system. The process and method of offline training are shown in §4.3.

State & Action. We use the metrics passed by Monitor module as the state of RL system, which is used to select the optimal CCA. We use normalized metrics instead of the exact values. This avoids exaggerating the impact of an input metric with very large values on the final model. Further, normalization helps Selector module generalize the network conditions it observes during the training phase to unseen network conditions and achieve better performance. As for the action, the RL system uses CCA candidates as the possible action values.

Reward. Reward is an important factor affecting the RL system's performance. Specifically, the rewards the RL agent gains at each step quantify its performance to improve its subsequent action. Numerous RL-based congestion control-related solutions adopt Power or its variants as reward [7, 28, 31, 42, 57]. However, our experiment results show that application-layer metrics are more appropriate reward indices than QoS-oriented transport layer metrics (§2.1). Therefore, we directly utilize the gradient of QoE as the reward for our RL system.

We regard the RCT as the QoE of web services, which is also a common QoE choice for such services. RCT is the time taken from sending the request to receiving the last byte of the response, recorded on the client side. RCT mainly includes the transmission elapsed time within the network (transmission time) and the response queuing time on the sender (i.e., Qws in Tab. 2). For web services, we employ this value as the RL system reward for the following reasons:

- When the sending rate is high, the response queuing time is much smaller than the transmission time. Therefore, RCT is approximately equal to the transmission time, which reflects the current transmission efficiency, and thus can evaluate the current action (i.e., CCA).
- When the sending rate is low, or is lower than the delivery rate from the sender application layer to the sender transport layer, RCT mainly includes the response queuing time. In this case, RCT reflects the growth and drain of the queue within the send buffer. If the current action is better than the previous action, it will suppress the queue growth or accelerate the queue draining. The change of the queue will be reflected in the change of RCT, and will further evaluate the merit of actions in one training round.

Specifically, we use the gradient of RCT as the reward: $R = \ln \frac{Last\ RCT_{avg}}{Current\ RCT_{avg}}$. In each step, we record the average RCT in time units and compare it with that of the last step. After one entire training episode, we normalize all the rewards uniformly.

Learning algorithm. Floo adopts actor-critic RL, and is trained using the Proximal Policy Optimization (PPO) algorithm [45]. PPO is an advanced RL algorithm that is adept at exploring policies with continuous features. PPO addresses the issues of the traditional policy gradient philosophy and improves the utilization of data and the model stability by the design of importance sampling and clipping. Appendix A details how PPO is utilized in Floo.

3.4 CCA State Migration

To deal with the challenges described in §2.3, our design goals are: (1) Inherit the network path estimation to speed up CCA convergence and avoid performance degradation. (2) Retain the characteristics of new CCAs consistent with the original design goals. In our design, the CCA state migration mechanism considers all the CCA-related variables, i.e., the CCAs' phase and the variables used in CCAs. This mechanism can be applied to non-learning CCAs, e.g. Cubic, BBR and Copa.

CCA Phase Migration. CCA phase migration is concerned with the state transition within the CCAs. With packet loss no longer the only congestion signal, the emerging CCA phases become more complex. However, a common feature of non-learning CCAs is that they all probe the path and estimate

Variable Type	Description
Sending rate variables	Variables that directly determine the sending rate, e.g. CWND, Pacing rate, etc.
Observation variables	Observations of the connection, e.g. smooth RTT, max delivery rate, etc.
Parameter variables	Variables related to CCA design, e.g. β=0.8 in Cubic when packet loss.
Other variables	Variables that maintain CCA's current state, e.g. *fulled_pipe* in BBR, and *velocity* in Copa.

Table 3: Four types of CCA-related variables.

their occupancy of the path based on the feedback. Therefore, we coarsely classify CCA phases into two categories based on how well the CCA probes the path.

The first category is the non-converged phase, i.e., where CCA has not formed a complete awareness of the path or does not fully utilize the available capacity. The non-converged phase includes both the slow start phase and the situation where the CCA would not fill the pipe for other purposes, such as ProbeRTT in BBR. The second category is the converged phase, i.e., where CCA adjusts the sending behavior based on the observations of the path after the slow start, including the congestion avoidance phase of traditional CCAs, the ProbeBW phase of BBR, and the moving phase of Copa.

We adopt different measures to migrate different phases. If the switching happens at the converged phase of the old CCA, Floo makes the new CCA directly enter the converged phase.This avoids massive packet loss caused by the slow start phase after switching. We do not perform switching at the non-converged phase. On the one hand, the statistics collected during the non-converged phase are unreliable. On the other hand, the non-converged phase usually does not last too long, so it will not cause much damage even if the switch is not made immediately. Note that there is one exception: if the new CCA is BBR, Floo makes the BBR enter the ProbeRTT phase first. Though the ProbeRTT phase is not converged, it will affect the performance of the converged phase.

CCA Variable Migration. CCA variable migration is mapping variables from the prior CCA to the new CCA. The variables maintained by various CCAs are different and affect CCA switching performance differently. Therefore, we group all variables into four types, as shown in Tab. 3. According to our two design goals, we adopt the corresponding migration methods for each type.

- **Sending rate variables.** Sending rate variables, such as CWND and pacing rate, directly determine the sending rate. Therefore, they need to inherit the prior rate, thus ensuring smooth switching. The key issue here is the conversion between the rate-based CCAs and window-based CCAs. We use the relationship that $CWND = pacing\ rate * RTT$ to calculate the migrated values.
- **Observation variables.** Observation variables are estimated statistics for the network path. The observation variables collected in the converged phase can directly follow the new CCA. Considering that some important observation variables are not preserved by all CCAs, we additionally preserve the bottleneck bandwidth and minimum RTT at the granularity of the connection.
- **Parameter variables.** Parameter variables are related to the design of the CCA, which are basically fixed values and will determine the performance. Therefore, we do not perform any manipulation on these variables.
- **Other variables.** Other variables maintain CCA's current states, most of which are computed from observation variables. We migrate them based on the phase migration methods. The left ones are simply initialized to default values.

4 Implementation and Training

We implement Floo atop QUIC in the production environment of Dianping service with O(10M) daily active users. We first introduce the implementation of Floo (§4.1). Then, in order to make our model applicable to real applications, we conduct a large scale passive measurement on Dianping application from Meituan, analyze the traffic patterns of the application and use the collected application traces for training (§4.2). Also, we train the RL-based CCA selection model with the numerous newly collected real application traces and wireless network traces (§4.3) to make Floo Selector module suitable for real application scenarios.

4.1 Implementation

We implement Floo based on QUIC [29][2] in user space. Floo only requires modification on the sender side. For the training phase, we implement Floo's RL-Agent on top of Tensorflow [6]. After the training phase, we obtain the trained model, which is used in Floo's Selector module. The training phase is well presented in §4.3.

As for the applications, we slightly modify Dianping to support Monitor module of Floo, then the modified Dianping can run on top of Floo. We use it in real-world experiments (§5.3). Additionally, we also implement a simple request-response messaging application (Application S) atop Floo, which is used in the training phase of Floo and all the emulated experiments in §5. Application S generates requests and responses according to the application traces introduced in §4.3.1. In the training phase of Floo, Application S negotiates with RL agent about the information of state, action, reward, etc. For all the emulated experiments in §5, Application S employs the well-trained model. The sender of Application S selects CCA dynamically according to Floo's Selector module.

CCA Candidates. We consider the CCAs that have been deployed in real Internet environments as candidates for our CCA selection policy. Firstly, for deployability, we mainly consider widely-deployed CCAs, and thus choose the most two widely-deployed [36] CCAs, Cubic and BBR. Secondly, for effectiveness, CCA candidates should cover diverse QoS metrics. Therefore, we also use the loss-resilient Westwood

[2]We use an IETF QUIC implementation, ngtcp2 [2].

and latency-sensitive Copa. These four CCAs have different preferences for QoS targets (§2.1).

4.2 Application Dataset

We measure the traffic patterns of a real mobile application in production environments. These measurements illuminate the nature of request-response messaging traffic and provide the basis for constructing CCA selection policies that can be used for real applications.

We perform a large-scale passive measurement on Dianping from Meituan. Users can make purchases through this application, and the main user actions include searching, viewing images, etc. When the users are using the application, the client establishes a persistent connection with a frontend server, through which the application sends requests to the frontend server. We instrument the mobile APP client of that application, and after each request is completed, the instrumentation collects application-level logs and connection-level logs describing the finished transport process. We collected the logs of about 35 million request-response messages over two weeks.

First, we found that the connections of the application are persistent and would last 206s on average, which is much longer than the SP, supporting Floo's CCA selection. Then, we present the characteristics of the mobile web service, i.e. size and frequency of requests and responses sent through the persistent connections between the client APP and the frontend servers in Fig. 5. Fig. 5a shows the CDF of the request size. As we can see, over 80% of the requests are less than 10 KB, indicating that most of the upstream traffic is small and generally not the performance bottleneck. Fig. 5b shows the CDF of the response size. The responses have a diverse mix of small and large sizes with heavy-tailed characteristics. For the response workload, more than 70% of the responses are less than 10KB, but more than 60% of all bytes are in the 3.4% of responses.

Fig. 5c shows the time interval between the two consecutive sending of requests from the client. The inter-sending time between requests reflects the density and diversity of requests initiated by the application. Since this interval is influenced by both user behavior and application characteristics, we filter out the request initiation due to the user behavior. Specifically, the two requests with an inter-sending time greater than 1 second are considered two clicks of the user behavior. After filtering, as shown in Fig. 5c, 80% of the request inter-sending intervals are less than 44ms, and 38.4% of them are concurrent (0ms). *Therefore, although most of the requests and responses are tens to hundreds of kilobytes in size, the bandwidth needs of the application are still high.*

4.3 Training

Floo's training goal is to learn one policy that can select an appropriate CCA to achieve good QoE in diverse network environments. That policy should be applicable to real applications over real network environments. For this purpose, we use real application traces and real wireless traces for training in a controlled emulated environment (§4.3.1). Further, we use real QUIC implementations and Application S, instead of network simulators (§4.3.2). This allows the RL agent to have an experience close to that in real-world scenarios.

Parameter	Value Range (Min - Max)
RTT(ms)	10 - 50, 50 - 100, 100 - 150, 150 - 300
RTT Jitter / RTT	0 - 0.2, Jitter max = 20ms
Loss rate(%)	0 - 0, 0 - 0.1, 0.1 - 5
Buffer / BDP	0.3 - 0.9, 0.9 - 1.1, 1.1 - 1.5

Table 4: Network condition parameters.

4.3.1 Trace and Training Settings

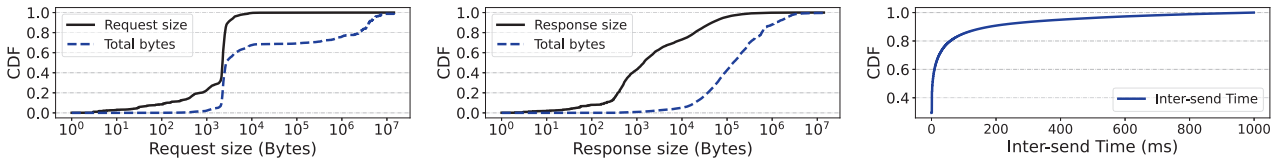
Application traces. We generate training application traces based on the distribution of the statistics collected from the measurements(§4.2). Specifically, we generate each request and response based on the CDF of the request and response size (Fig. 5a and 5b). The sending time of each request is determined based on the CDF of the inter-sending intervals (Fig. 5c). The server of Application S generates a corresponding response and delivers it down to the transport layer immediately after receiving a request. We train the RL model with many episodes, and each episode lasts 10 minutes. We generate a separate application trace for each training episode.

Network condition parameters. We use Mahimahi to emulate network paths and Traffic Control (TC) [8] to emulate RTT jitter. We adopt the network traces collected and used in previous works [3, 7, 30, 34, 38, 39, 47, 50], as listed in Tab. 6. These traces are employed to emulate the time-varying network path rate upper limit. They can be used to emulate various network conditions including 4G and 5G in both stationary and mobile scenarios.

Besides rate upper limit, RTT, RTT jitter, packet loss rate, and buffer size are also common network condition parameters [17, 52]. We select the values of these parameters using the space-filling WSP algorithm [17, 44] over the ranges listed in Tab. 4. WSP algorithm could generate multiple sets of network conditions based on the range of each parameter in order to emulate network conditions as diverse as possible. We generate 20,000 sets of network condition parameters. At the beginning of each training episode, we randomly select a network trace with one set of network condition parameters.

4.3.2 Training Method

We construct a training architecture consisting of learning agents, Application S clients and Application S servers. The client and server connecting to the same agent also establish a QUIC connection through Mahimahi. We set the episode to 10 minutes, which is long enough for CCAs to converge. For each training episode, the agent selects and configures the network condition parameters of Mahimahi and the application traces to be applied, as described in §4.3.1. The detailed training method is depicted in Appendix C.



(a) Distribution of the request size. **(b)** Distribution of the response size. **(c)** Distribution of the inter-sending interval.

Figure 5: Distributions of request and response characteristics from the measurements depicted in §4.2.

5 Evaluation

We first introduce our experimental setup in §5.1. We then evaluate Floo in the following aspects:

- **Consistent high performance.** We evaluate Floo over different scenarios. Evaluation shows that Floo achieves the highest throughput and lowest delay under different network conditions. Floo can reduce the application RCT by up to 52.7% on average, and up to 78.16% at the tail (§5.2).
- **Performance in the real world.** We implement Floo in a popular mobile web service and measure the performance for 96 hours. Experiments with real users show that Floo can reduce RCT by about 14.26% in the real world (§5.3).
- **Overhead.** Floo has acceptable overhead, with about 1.4% additional CPU utilization and sub-ms magnitude of additional time consumption (§5.4).
- **Improvement deep dive.** Finally, we evaluate the effectiveness of the design of Floo (§5.5).

5.1 Setup

We evaluate Floo in both emulated networks (§5.2, §5.4, §5.5) and large scale production environment (§5.3).

Emulated environment. We evaluate Floo in a controlled environment by emulating different network conditions with Mahimahi and generating new application traces. In our testbed evaluation, we implement Application S atop Floo, which sends requests and responses with application traces, and collects statistics for evaluation. We conduct experiments under 60 scenarios, including 10 stationary WiFi traces, 20 stationary cellular traces, and 30 mobile cellular traces. We also use the WSP algorithm to select 60 sets of other parameters, using the same method as §4.3.1, and importantly, the combinations of trace and parameter sets are different from those traces used in training. We compare the performance of Floo respectively with Cubic, BBR, Copa, Westwood and Vivace [19]. In each scenario, we send requests and receive responses using different algorithms for 3 minutes with the newly generated application traces.

Large scale production environment. We implement Floo in Dianping, with O(10M) daily active users. Our experiments are conducted in production environment where clients are heterogeneous including different OS, HTTP versions, etc. We manually enable Floo for a fraction of users, measure the performance for four days and collect 35 million request logs. In the experiments, we set the selection period (SP) as 12s. A further analysis of the impact of different SP values is detailed in Appendix F.

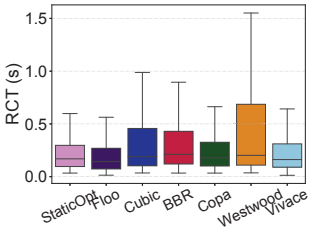


Figure 6: Floo achieves lower RCT than QoS-oriented CCAs.

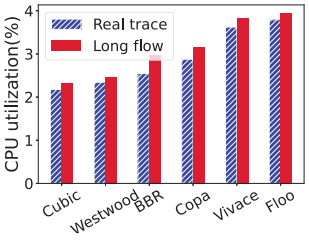


Figure 7: CPU utilization.

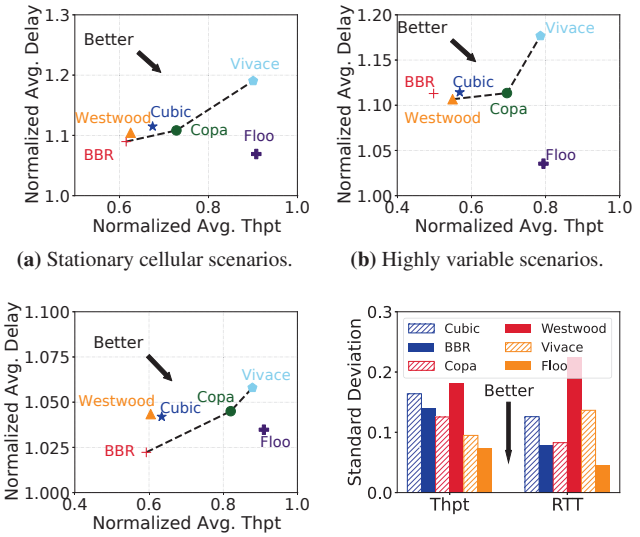
5.2 Consistent high performance

Here, we demonstrate that Floo achieves consistent high performance over different network scenarios.

Overall performance. We evaluate Floo under different scenarios, with 60 real-world traces. We record the RCT and show the performance of all scenarios in Fig. 6. For each scenario, we compare the four CCA candidates, and select the optimal CCA which achieves the lowest average RCT. The aggregated best choices for all scenarios is presented as Static_Opt. Floo achieves the lowest RCT, and reduces the overall RCT by a median of 20.11% to 32.54% and 21.18% to 78.16% in the 90^{th} percentile. The average RCT was reduced by 14.3% to 52.7% compared with QoS-oriented CCAs.

Remark 1 (Cubic and Westwood): Floo reduces the average RCT by 52.7% compared to Cubic and 50.8% to Westwood. For the 90 percentile (the tail) RCT, Floo shows great improvement, and has a 74.6% reduction compared to Cubic and 78.18% to Westwood. This is because that different types of CCA have different scopes of application. Empirically speaking, the performance of loss-based CCAs (i.e. Cubic and Westwood) degrades with high RTT and random packet loss, and will suffer longer tail latency. The improvement of Floo in the tail RCT demonstrates its selection accuracy to not use Cubic/Westwood when the network condition is poor.

Remark 2 (Copa and BBR): Compared to Copa and BBR, Floo reduces the average RCT by 20.53% and 14.3% respectively. For the 90 percentile (the tail) completion time, Floo has a 21.18% reduction compared to Copa and 21.57% to BBR. The improvement of Floo over the four CCA candidates validates the accuracy of our Selector module.



(a) Stationary cellular scenarios. **(b)** Highly variable scenarios.

(c) Wifi scenarios. **(d)** SD. of all scenarios.

Figure 8: A set of transport layer results for the three scenarios. (a)(b)(c) present the normalized avg. delay, and avg. throughput, where the dashed line represents the Pareto front of baselines. (d) presents the standard deviations of the overall results.

Remark 3 (Static_Opt): Floo can adapt to variable and dynamic network scenarios, and switch to a better CCA during the connection whenever the network condition changes. Therefore, Floo obtains better performance than statically optimal selection (9.6% reduction in avg. RCT). The advantage of Floo over Static_Opt demonstrates the need for CCA switching during the connection.

Remark 4 (Vivace): Vivace is also designed for performance. Therefore, we adopt Vivace as a baseline to evaluate Floo's ability to improve application QoE. Results show that Floo achieves 17.72% lower RCT on average than Vivace, and 25% reduction in the 90^{th} RCT. This is because Vivace still focuses on transport layer metrics and the utility function of Vivace is not consistent with the QoE. In addition, the penalty for packet loss and latency in the utility function makes Vivace less resistant to random packet loss.

Transport layer performance under different scenarios. In our emulated experiments, we consider three scenarios, including stationary cellular scenario, highly variable scenario and unseen WiFi scenario. We analyze the transport layer metrics under different scenarios. We consider two performance metrics: average smooth RTT and average throughput. For each scenario, we normalize the RTT and throughput performance of all CCAs (including Floo) to the minimum delay and maximum throughput achieved on that scenario, respectively. Then, we average all normalized values over all scenarios and show the results in Fig. 8. More detailed results are presented in Appendix E.

- **Stationary cellular scenarios.** In our evaluations, there are 20 traces of the stationary cellular network, including indoor [34] and outdoor [7] traces (Fig. 8a).
- **Highly variable scenarios - mobile cellular.** Similarly, we tested 30 mobile cellular traces, which are highly variable scenarios (Fig. 8b). These traces are collected when walking and driving under 4G [7] and 5G mmWave [38]. We also adopt the 4G measurements on high-speed rails [30] to construct a scenario with violently fluctuating bandwidths.
- **Unseen scenarios - WiFi.** To evaluate the behavior of Floo in unseen scenarios (Fig. 8c), we use WiFi traces that have not been employed in the training, We used 10 WiFi traces from [35], including traces from office and a public WiFi provided by a crowded restaurant during dinner hours.

Results show that Floo generally achieves the highest throughput with the lowest latency under different scenarios. Even in unseen scenarios, Floo shows advantages, demonstrating Floo's generalization capability. The improvement of Floo demonstrates that, besides QoE improvements, directly optimizing the application QoE through CCA selection approach can further improve transport layer capabilities.

We also present the stability of throughput and RTT (i.e., the standard deviation of the normalized average throughput and RTT) of each CCA under all scenarios in Fig. 8d. Results show that Floo could almost achieve stable high performance in all three scenarios, especially in terms of latency.

5.3 Real-world performance

We implement Floo in Dianping with O(10M) daily active users. We manually enable Floo for a fraction of users. Specifically, we deployed Floo on the front-end server to serve the persistent connection between the front-end server and the client. We enabled Floo for 5% of the users for evaluation. Besides Floo, we also implement a Floo with Part Switching (P-Floo), enable P-Floo for another 5% users and evaluate the effectiveness of switching algorithms. As a comparison, we set up another 5% of the users to use Cubic, BBR, Copa, Westwood and Vivace respectively. We collect logs for 96 hours, resulting in more than 35 million request logs, covering users from more than 50 countries and regions. We collect RCT from client side and the results are shown in Fig. 9 and Fig. 10.

Floo is still able to reduce the RCT and obtain optimal performance in real-world scenarios. Floo achieves a QoE improvement of 8.07% to 14.26% in real scenarios, with a reduction of about 25.5% for tail RCT. The difference in RCT between the real scenario and the emulated evaluation mainly comes from the different distribution of network conditions. For emulated evaluation, we aim to cover various network conditions by selecting as diverse network environments as possible. In contrast, the network states in the real scenario are not uniformly distributed. We found that there are about 57.89% of scenarios are under better network conditions (i.e., packet loss rate is 0% and min RTT is less than 44ms). Therefore, unlike the significant advantage of Copa in the emulated

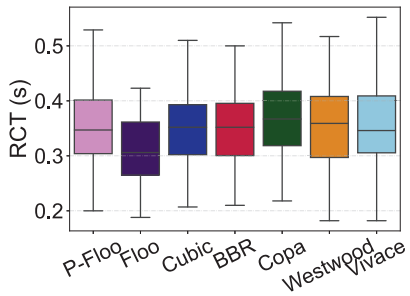


Figure 9: In real world experiments, Floo brought 12.9% reduction on average RCT.

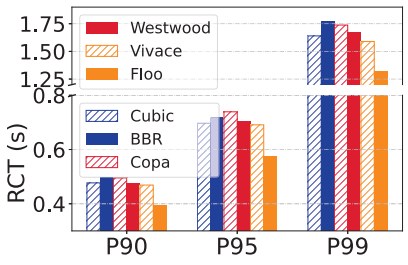


Figure 10: Floo reduces 25.5% of the 99^{th} percentile (the agestail) completion time.

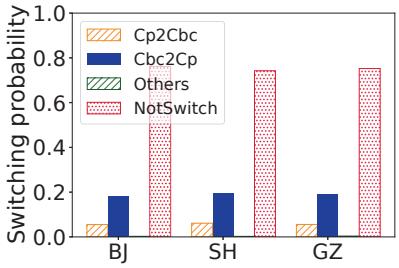


Figure 11: The probability of switching types of different user groups.

Part	CPU utilization (%)	Time Consumed (μs)
Network Monitor	0.784 (3.109)	47.4375
Selector	0.431 (3.540)	66.1875
Switcher	0.233 (3.773)	19.5625

Table 5: CPU overhead and time consuming of Floo.

scenario, Cubic and Westwood obtains a lower average RCT in the real world. As a result, the improvement brought by Floo in real world is a little lower.

We count the switching frequency and types in the real world in Fig. 11. With the SP of 12s, the probability of a CCA switching occurring at selecting is 25.76%, which implies an average switching interval of about 47s. Among the switching, the mutual switching between Copa (Cp) and Cubic (Cbc) is the most frequent, with more than 74% of the switching being Cubic to Copa and more than 23% being Copa to Cubic. We group users based on the location of CDN nodes they access. The results of switching frequency and actions in different groups are similar.

5.4 Overhead

We report the CPU utilization and runtime overhead of Floo.

CPU utilization. We measure the system overhead of Floo and compare it with other state-of-the-art CCAs. We performe experiments on an emulated network (with 48Mbps bottleneck link and 20ms RTT) for 6 minutes. We measure the average CPU utilization with real application traces and a long flow separately and show the results in Fig. 7. All algorithms are implemented atop QUIC in user space. Although Floo has a higher overhead compared to classical CCAs, however, compared to Cubic, which has the lowest overhead, the additional overhead is only 1.4%.

We measure the CPU utilization of each part by incremental experiments. Specifically, we separately measure the CPU utilization of only Monitor module, Monitor module with Selector module, and the complete Floo. We define the computed overhead of each part as the difference in CPU utilization (%) between two measurements. The results are shown in Tab. 5.

Time consuming. We show the time consumed by recording the time spent for each module. Tab. 5 presents the results taken as an average across 16 runs. We see that the additional consumed time introduced by Floo is at the sub-millisecond level, which is much less frequent than that of CCA selection (about 38.9s on average in our testbed experiments). Specifically, Selector module takes the most time (about 66μs) because of the complex calculations for CCA selection. Monitor module also consumes about 47μs to collect the additional information. Switcher module executes the state migration mechanism, which consumes about 19us.

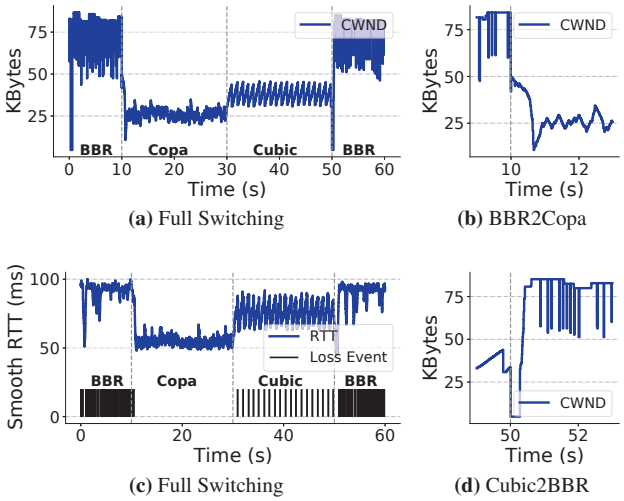
5.5 Floo deep dive

Here, we evaluate the effectiveness of Floo's design of state migration, generalizability to other QoE metrics and resilience to stochastic packet loss.

5.5.1 Effectiveness of state migration

Functional validation. Floo encounters situations where the path conditions change during transmission and switching-on-fly is required. To evaluate the effectiveness of Floo's state migration algorithm, we manually set the CCA switching every 20s and switch between all CCAs in an emulated environment. We compare Full Switching and Part Switching under 60 different scenarios.

Fig. 12 shows the details of the congestion control switching process. We show the performance of switching from BBR to Copa, and to Cubic. We do not show additional details of Westwood, since Westwood is basically similar to Cubic in terms of algorithm design. As described in §2.3, packet loss occurs under Part Switching when switching occurs without convergence. In addition, Copa maintains a high CWND and thus experiences a high RTT with packet loss due to the distortion of estimation of path conditions. Full Switching, as shown in Fig. 12, avoids these problems and maintains the CCA characteristics consistent with their design. Specifically, when switching to Copa (Fig. 12b), Floo is able to decrease the CWND within 1 second, thus quickly emptying the queue built by BBR and maintaining low latency. When switching to BBR (Fig. 12d), Floo first enters the ProbeRTT phase so as to obtain the RTProp, which the prior CCA did not maintain earlier. After that, BBR does not enter the Startup phase, but gradually converges with the ProbeBW phase.



(a) Full Switching **(b)** BBR2Copa

(c) Full Switching **(d)** Cubic2BBR

Figure 12: Details of Full Switching. (a)(c) Full Switching statistics of CWND, smooth RTT and loss event. (b) Detailed CWND of Full Switching from BBR to Copa. (d) Detailed CWND of Full Switching from Cubic to BBR.

Performance analysis. We compare the performance of Full Switching and Part Switching in both emulated and production environments. Fig. 13 shows the distribution of packet loss rate within 1 second after the switching in the emulated environment. Part Switching leads to more packet loss, while Full Switching significantly reduces the packet loss rate, especially when switching from Cubic to the low-latency algorithms, i.e. BBR and Copa. For real-world experiments, Fig. 9 shows that the average RCT is reduced by about 7.59% with Full Switching compared to Part Switching.

5.5.2 Generalizability to other QoE metrics

We change QoE to tail RCT in order to analyze Floo's ability to generalize to other QoE metrics. Accordingly, we set Reward as $R = \ln \frac{Last\ RCT_{p90}}{Current\ RCT_{p90}}$ and retrain a new RL model. We evaluate Floo-P90 under the same 60 scenarios as §5.2. For each scenario, we record the RCT of total requests, the RCT at the 50th percentile, and the RCT at the 90th percentile[3], respectively. We gather the value of all scenarios in Fig. 14.

Floo-P90 is not as good as Floo in terms of total performance of all requests. However, for the 90th percentile RCT, Floo-P90 has a significant improvement, with a reduction of 21.78% on average. Compared to Floo, Floo-P90 selects Cubic less frequently by 6.03%, while the frequency of using BBR, Copa and Westwood increased by 4.02%, 1.0% and 1.0%, respectively. This is because the loss caused by Cubic when filling the buffer can result in a long RCT, while BBR are relatively conservative. As a comparison, Floo reduces RCT at the 50th percentile by about 19.49% compared to Floo-P90. The above results show that with our mechanism, there can be a significant improvement on the target QoE metric. See §7 for more analysis on generalization ability.

[3]We do not compare the 90th percentile RCT of all secnarios because it represents the performance under poor network scenarios.

5.5.3 Resilience to stochastic packet loss

We also analyze Floo's resilience to stochastic packet loss. Stochastic packet loss often occurs under cellular networks due to channel interference, mobility, etc [24, 51]. We evaluate the performance of Floo with a single flow on a link with 4 Mbps bandwidth, 20 ms RTT, 10 KB buffer, and varying random loss rate from 0% to 10%. As shown in Fig. 15, Floo still maintains a low RCT when the stochastic loss rate is set to 10%. It is worth mentioning that Vivace maintains a low RCT until a loss rate of about 4%. After that, corresponding to the 5% loss resistance in the utility function [19], the average RCTs increase dramatically, even up to 9.5 times of the no packet loss case. In addition, the performance of Vivace suffers uncertainty and instability with random packet loss.

6 Related Work

QoE-oriented transport-layer optimization. There are many other ways to conduct QoE-oriented transport optimization [18, 19, 33], while conveying QoE to transport layer by CCA selection is more appropriate. QoS, the target of transport optimization, is reflected in the behaviors in the network, e.g. how to utilize the bottleneck queue, where CCA is the most effective procedure to control. For example, better packet scheduling could improve the host queueing time through reordering the packets [16]. However, packets from one application always have the same QoE, leaving little optimization space for packet scheduling. And flow control schemes could also decide the sending rate, while it is not aware of network behaviors. Therefore, conveying QoE through CCAs is more appropriate.

QoE-oriented CCAs. We are not the first to observe that CCAs should be optimized towards application QoE. One line of solution is to integrate application design and the transport layer behavior [13, 21]. However, these works are designed for specific applications and redesign is needed if designers want to migrate their good performance to other applications. There are also proposals to use application requirements, such as deadline [55] and priorities [56], to guide the design of CCA at the transport layer. However, they can only be used for application requirements that can be directly understood by the transport layer. For example, deadline can be identified as the data delivery time, which the transport layer can estimate and optimize directly from RTT and packet loss events. For the complex QoE metrics, a possible solution is to adopt mature algorithms, such as reinforcement learning, yet we found it impractical. If we put translated QoS as the goal of RL [7, 26], the gap between QoE and QoS remains. If we put QoE as the goal of the RL, the indirect and distant connection between QoE and cwnd/rate decisions makes the training extremely hard to converge. RL-based CCAs also

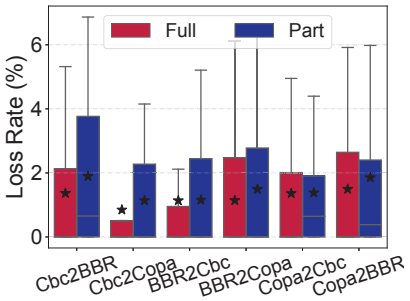


Figure 13: Loss rate of Full Switching and Part Switching within 1s after the switching. Cbc represents Cubic.

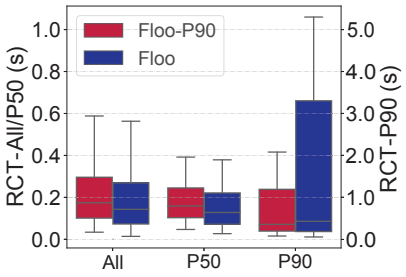


Figure 14: Tail-RCT-oriented Floo-P90 significantly reduces the 90th percentile RCT by 21.78% on average.

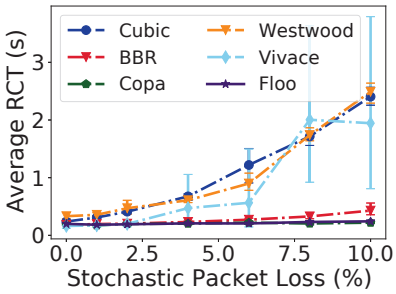


Figure 15: Impact of stochastic loss on RCT.

have interpretability issues in the wild. Therefore, Floo adopts CCA selection mechanism to bridge the gap between the complex, application-oriented QoE and the direct, transport capability-oriented QoS.

CCA-selection methods. There has been some work that adopts CCA selection methods [15,20,42,57]. Existing works still aim to optimize transport layer performance. However, as stated in §2.1, QoS-driven CCA-selection methods will suffer performance degradation when selection criteria are not entirely consistent with QoE. In addition, existing schemes cannot address the two challenges (§2.3) well. Their selection policy generation methods do not consider QoE, and they could not support seamless switching either.

Advanced CCAs for mobile web service. In recent years, many advanced CCA schemes have been proposed, including CCAs specifically designed for wireless network, such as Sprout [50], Verus [53], and emerging learning-based CCAs [7, 19]. However, these CCAs are still oriented to optimize the transport layer performance and do not address the QoS and QoE mismatch. Moreover, considering the practical issues [7, 15] and unproven performance in production environments, we did not consider them as CCA candidates.

7 Discussion

Generalization to heterogeneous applications and scalability to various CCAs. In this paper, we propose a solution for mobile web service, aiming to reduce RCT, and using four classical CCAs as candidates. In fact, Floo could be applied to heterogeneous applications and various CCAs. For example, for streaming applications, Floo can be reused by considering a chunk as a request. For complex QoE, as long as we could extract the traces, characteristics, and the QoE metric of the application, Floo can theoretically be used for any application without any idea of the implementation details and optimization techniques of the application. For CCAs, Floo can incorporate various CCAs into the selection mechanism. Nevertheless, CCA state migration has to be considered. Our design in §3.4 can be extended to all non-learning algorithms. However, generalizing to complex applications and switching between complex CCAs are not designed and verified in this paper, and are future work.

Fairness and friendliness. Floo selects among various CCAs, and the fairness and friendliness of Floo is consistent with that of the CCA candidates. In this paper, we select from the deployed CCAs, which already has had theoretical and experimental analysis of fairness and friendliness [9, 11, 48].

Portability to TCP in Linux kernel. Although Floo is implemented atop QUIC, Floo can still be applied to TCP implementation in Linux kernel. Firstly, eBPF technique [1, 10] provides a safe and convenient way to interact between user space and kernel. One can imagine that Floo works in user space, extracts information from the kernel and delivered the selected CCA to the kernel. The state migration mechanism, on the other hand, requires further modifications to the kernel. Secondly, as for the integration with mechanisms specific for TCP or QUIC, e.g., multi-streaming in QUIC, Floo is orthogonal to pre-CCA optimizations. Therefore, for implementing Floo over TCP without pre-CCA optimizations, Floo can also select the appropriate CCAs in respective situations.

8 Conclusion

We propose Floo, a QoE-oriented CCA selection mechanism for mobile web service. Floo uses QoE as the selection criterion and employs RL techniques to construct the mapping from the transport layer and application layer metrics to CCAs. Floo switches smoothly during the transmission. We implement Floo in a popular mobile web service, and evaluate Floo in both emulated and production environments. Experiments show that Floo reduces the RCT by 14.3% to 52.7% in different scenarios.

This work does not raise any ethical issues.

Acknowledgements

We sincerely thank our shepherd Wei Gao, anonymous reviewers, and labmates in Routing Group from Tsinghua University for their valuable feedback. The research was supported by the National Natural Science Foundation of China under Grant 62002192, Grant 62221003 and Meituan. Mingwei Xu and Enhuan Dong are the corresponding authors of the paper.

References

[1] ebpf. https://ebpf.io/what-is-ebpf/.

[2] ngtcp2. https://github.com/ngtcp2/ngtcp2.

[3] Raw data - measuring broadband america. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america.

[4] Desktop vs Mobile vs Tablet Market Share Worldwide. https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet, 2022. Accessed: 2022-8-19.

[5] OpenAI-baseline. https://github.com/openai/baselines/tree/master/baselines/ppo2, 2022. Accessed: 2022-8-29.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[7] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. 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*, pages 632–647, 2020.

[8] Werner Almesberger et al. Linux network traffic control—implementation overview, 1999.

[9] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *Proc. USENIX NSDI*, pages 329–342, 2018.

[10] Lawrence Brakmo. Tcp-bpf: Programmatically tuning tcp behavior through bpf.

[11] Yi Cao, Arpit Jain, Kriti Sharma, Aruna Balasubramanian, and Anshul Gandhi. When to use and when not to use bbr: An empirical analysis and evaluation study. In *Proceedings of the Internet Measurement Conference*, pages 130–136, 2019.

[12] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

[13] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.

[14] Claudio Casetti, Mario Gerla, Saverio Mascolo, Medy Y Sanadidi, and Ren Wang. Tcp westwood: end-to-end congestion control for wired/wireless networks. *Wireless Networks*, 8(5):467–479, 2002.

[15] Kefan Chen, Danfeng Shan, Xiaohui Luo, Tong Zhang, Yajun Yang, and Fengyuan Ren. One rein to rule them all: A framework for datacenter-to-user congestion control. In *4th Asia-Pacific Workshop on Networking*, pages 44–51, 2020.

[16] Yong Cui, Chuan Ma, Hang Shi, Kai Zheng, and Wei Wang. Deadline-aware Transport Protocol. Internet-Draft draft-shi-quic-dtp-06, Internet Engineering Task Force, July 2022. Work in Progress.

[17] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies*, pages 160–166, 2017.

[18] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *Proc. USENIX NSDI*, pages 395–408, 2015.

[19] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *Proc. USENIX NSDI*, pages 343–356, 2018.

[20] Zhuoxuan Du, Jiaqi Zheng, Hebin Yu, Lingtao Kong, and Guihai Chen. A unified congestion control framework for diverse application preferences and network conditions. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 282–296, 2021.

[21] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *Proc. USENIX NSDI*, pages 267–282, 2018.

[22] Alfred Giessler, J Haenle, Andreas König, and E Pade. Free buffer allocation—an investigation by simulation. *Computer Networks (1976)*, 2(3):191–208, 1978.

[23] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[24] Habtegebreil Haile, Karl-Johan Grinnemo, Simone Ferlin, Per Hurtig, and Anna Brunstrom. End-to-end congestion control approaches for high throughput and low delay in 4g/5g cellular networks. *Computer Networks*, 186:107692, 2021.

[25] Yongkai Huo, Cornelius Hellge, Thomas Wiegand, and Lajos Hanzo. A tutorial and review on inter-layer fec coded layered video streaming. *IEEE Communications Surveys & Tutorials*, 2015.

[26] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019.

[27] Leonard Kleinrock. On flow control in computer networks. In *Proceedings of the International Conference on Communications*, volume 2, pages 27–2, 1978.

[28] Leonard Kleinrock. Internet congestion control using the power metric: Keep the pipe just full, but no fuller. *Ad hoc networks*, 80:142–157, 2018.

[29] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proc. ACM SIGCOMM*, pages 183–196, 2017.

[30] Li Li, Ke Xu, Tong Li, Kai Zheng, Chunyi Peng, Dan Wang, Xiangxiang Wang, Meng Shen, and Rashid Mijumbi. A measurement study on multi-path tcp with multiple cellular carriers on high speed rails. In *Proc. ACM SIGCOMM*, pages 161–175, 2018.

[31] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proc. ACM SIGCOMM*, pages 15–30, 2020.

[32] Michael Luby, Lorenzo Vicisano, Jim Gemmell, Luigi Rizzo, M Handley, and Jon Crowcroft. Forward error correction (fec) building block. Technical report, 2002.

[33] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. In *Proc. ACM SIGCOMM*, pages 615–631, 2020.

[34] Zili Meng, Yaning Guo, Yixin Shen, Jing Chen, Chao Zhou, Minhu Wang, Jia Zhang, Mingwei Xu, Chen Sun, and Hongxin Hu. Practically deploying heavyweight adaptive bitrate algorithms with teacher-student learning. *IEEE/ACM Transactions on Networking*, 29(2):723–736, 2021.

[35] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *Proc. ACM SIGCOMM*, pages 193–206, 2022.

[36] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. The great internet tcp congestion control census. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–24, 2019.

[37] Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. Pruning edge research with latency shears. In *ACM HotNets*, 2020.

[38] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand AK Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM Internet Measurement Conference*, pages 176–193, 2020.

[39] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. A variegated look at 5g in the wild: performance, power, and qoe implications. In *Proc. ACM SIGCOMM*, pages 610–625, 2021.

[40] Ravi Netravali and James Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *USENIX NSDI*, 2018.

[41] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[42] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.

[43] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proc. ACM SIGCOMM*, 2017.

[44] Jenny Santiago, Magalie Claeys-Bruno, and Michelle Sergent. Construction of space-filling designs using wsp algorithm for high dimensional spaces. *Chemometrics and Intelligent Laboratory Systems*, 113:26–31, 2012.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[46] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[47] Jeroen Van Der Hooft, Stefano Petrangeli, Tim Wauters, Rafael Huysegems, Patrice Rondao Alface, Tom Bostoen, and Filip De Turck. Http/2-based adaptive streaming of hevc video over 4g/lte networks. *IEEE Commun. Letters*, pages 2177–2180, 2016.

[48] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling bbr's interactions with loss-based congestion control. In *Proceedings of the internet measurement conference*, pages 137–143, 2019.

[49] Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. Http/2 prioritization and its impact on web performance. In *World Wide Web Conference*, 2018.

[50] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proc. USENIX NSDI*, pages 459–471, 2013.

[51] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption. In *Proc. ACM SIGCOMM*, pages 479–494, 2020.

[52] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.

[53] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *Proc. ACM SIGCOMM*, pages 509–522, 2015.

[54] Jia Zhang, Enhuan Dong, Zili Meng, Yuan Yang, Mingwei Xu, Sijie Yang, Miao Zhang, and Yang Yue. Wisetrans: Adaptive transport protocol selection for mobile web service. In *Proceedings of the Web Conference 2021*, pages 284–294, 2021.

[55] Lei Zhang, Yong Cui, Junchen Pan, and Yong Jiang. Deadline-aware transmission control for real-time video streaming. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2021.

[56] Chao Zhou, Wenjun Wu, Dan Yang, Tianchi Huang, Liang Guo, and Bing Yu. Deadline and priority-aware congestion control for delay-sensitive multimedia streaming. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 4740–4744, 2021.

[57] Jianer Zhou, Xinyi Qiu, Zhenyu Li, Gareth Tyson, Qing Li, Jingpu Duan, and Yi Wang. Antelope: A framework for dynamic selection of congestion control algorithms. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

A Learning Algorithm of Floo

Floo's agent interacts with the environment, gets a series of trajectories ((state, action, reward) or (s, a, r) for short), and updates the policy according to the information of its interaction. Upon receiving state (s_t), Floo's agent needs to choose a corresponding action (a_t), i.e. one CCA, and will get the reward(r_t) in the next step. The policy that Floo's agent used to choose an action, is defined as the probability distribution of actions: $\pi(s_t, a_t) \rightarrow [0,1]$. $\pi(s_t, a_t)$ is the probability of taking action a_t in state s_t. Then the agent uses the PPO algorithm to update the parameter θ of the policy π. The PPO algorithm is optimized from the policy gradient methods [46], which estimates the gradient of the expected total reward by observing the trajectories obtained by following the policy. The gradient of Policy Gradient can be computed as:

$$\nabla_\theta J(\theta) = \mathrm{E}_{(s_t,a_t)\sim\pi_\theta}\left[\mathrm{A}^\theta(s_t,a_t)\nabla\log\pi_\theta(a_t^n \mid s_t^n)\right] \quad (1)$$

$\mathrm{A}^\theta(s_t,a_t)$ is the Advantage Function, which represents the difference in the expected total reward when we choose action a_t in state s_t, compared to the expected total reward for the action drawn from policy π_θ.

Policy Gradient is an on-policy method where the collected sample (s_t, a_t, r_t) is used only once. In order to make full use of the training data and improve the learning efficiency, PPO extends the Policy Gradient method. The original policy is denoted as π_θ, and when the gradient $\nabla_\theta J(\theta)$ is applied to the original policy π_θ, the new policy is denoted as $\pi_{\theta'}$. At this point, if we want to reuse the data generated by the policy π_θ to update $\pi_{\theta'}$, considering the different distributions of trajectories in $\pi_{\theta'}$ and π_θ, an importance sampling method is needed:

$$E_{x\sim p}[f(x)] = E_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right] \quad (2)$$

Therefore, the gradient of the off-policy is calculated as follows, with the parameters before and after the update denoted as θ and θ':

$$\nabla_\theta J(\theta) = \mathrm{E}_{(s_t,a_t)\sim\pi_{\theta'}}\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta'}(a_t \mid s_t)}\mathrm{A}^{\theta'}(s_t,a_t)\nabla\log\pi_\theta(a_t^n \mid s_t^n)\right] \quad (3)$$

Trace	Year	Type (Num)	Stationary / Mobile	Description
Lumos [38, 39]	2020	4G (166), 5G mmWave (121)	Stationary, Mobile (walking, driving)	Verizon's 4G and 5G service in Minneapolis.
NYC [7]	2019	4G LTE (23)	Stationary, Mobile (bus, taxi)	Cellular traces gathered in NYC.
PiTree [34]	2019	4G(61)	Stationary	Measurement of indoor 4G bandwidth.
HSR [30]	2018	4G (33)	Mobile (high-speed rails)	4G measurements on high-speed rails.
FCC18 [3]	2018	4G (397)	Stationary	The broadband network in 2018 provided by FCC.
Ghent [47]	2016	4G (40)	Mobile (foot, bicycle, bus, tram, train, car)	4G measurements in 2016 by Ghent University.

Table 6: The description of the real network traces.

The objective function is calculated as Eq. (4). To ensure that the difference between the policy before and after the update is not too large, PPO adds a constraint to the objective function. The clip function forces $\frac{\pi_{\theta(a_t|s_t)}}{\pi_{\theta'(a_t|s_t)}}$ between $1-\varepsilon$ and $1+\varepsilon$, and finally takes the minimum value among the rewards that have been clipped and those that have not been clipped.

$$J^{\theta'}(\theta) = \sum_{(s_t,a_t)} min(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta'}(a_t \mid s_t)} A^{\theta'}(s_t,a_t), \\ \text{clip}(\frac{\pi_{\theta(a_t|s_t)}}{\pi_{\theta'(a_t|s_t)}}, 1-\varepsilon, 1+\varepsilon) A^{\theta'}(s_t,a_t)) \tag{4}$$

The detailed derivation and sample code can be found in [5, 45].

B Real Network Traces

Tab. 6 shows the traces used in Section 4.3.1.

C Detailed Training Method

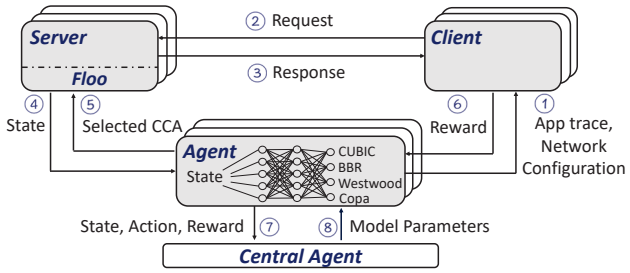


Figure 16: The workflow of the training phase of Floo. Multiple (Agent, Server, Client) sets run simultaneously.

We construct a training architecture as shown in Fig. 16. Each learning agent (Agent in Fig. 16) establishes two connections with an Application S client (Client in Fig. 16) and an Application S server (Server in Fig. 16) respectively, communicating about the experimental configurations and training trajectories. Each server/client only connects to one agent. The client and server connecting to the same agent also establish a QUIC connection through Mahimahi. For each training episode (10 min), the agent selects and configures the network condition parameters of Mahimahi and the application traces to be applied, as described in §4.3.1. During the interaction between the client and the server, the agent receives the states and rewards from the server and client respectively, and selects the corresponding action (i.e., CCA) based on the acquired state. The selected CCA is switched smoothly by Floo on the server.

To accelerate the training and improve the generalization performance of the RL model, we employ a distributed framework. We distribute 10 (Agent, Server, Client) sets. All the agents, servers, and clients are deployed in a cluster. These servers/clients are connected to the agents with high-speed links. Each agent observes a series of trajectories, and continuously sends the tuples (state, action, reward) to the central agent. The central agent then uses the PPO algorithm to compute the gradients (Eq. (3)) and updates the parameters in the selection model (Eq. (4)). The updated model will be pushed to each agent and will be used for the next episode. Tab. 7 in Appendix D shows the detailed model and parameters used during the training.

D Training Setting

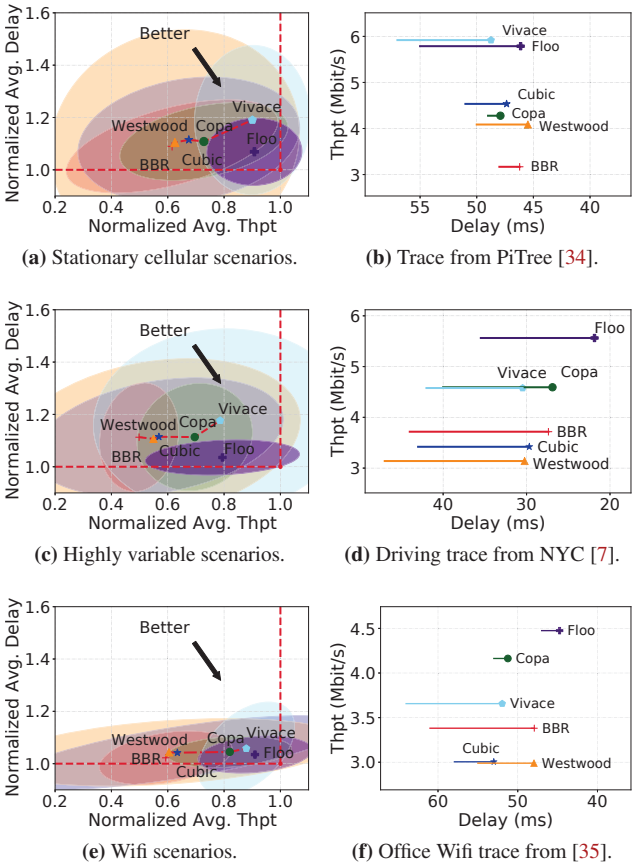
Floo uses an actor-critic architecture. Floo's actor, taking state and outputting action, use one hidden layer with 200 units. The output layer is a softmax layer to map to probabilities of actions. The critic networks, taking state and outputting V values, have one hidden layer with 200 units. The output layer is a linear unit representing the V function. All hidden layers in actor and critic networks are followed by Leaky ReLU nonlinearity. Tab. 7 shows other parameters used during the training of Floo.

Parameter	Value
Optimizer	Adam
Episode duration	10min
Actor's Learning Rate	0.0001
Critic's Learning Rate	0.0002
Discount Factor	0.99
ε in clip function	0.2

Table 7: Parameters used for the training.

E Detailed Results in Emulated Experiments

Our emulated experiments involve three scenarios: stationary cellular scenario, highly variable scenario and unseen WiFi scenario. We analyze the transport layer metrics, including average smooth RTT and average throughput. For each scenario, we normalize the RTT and throughput performance of all CCAs (including Floo) to the minimum delay and maximum throughput achieved on that scenario, respectively. We show the normalized results in Fig. 17a, Fig. 17c and Fig. 17e. The ellipse indicates the standard deviations from the average



(a) Stationary cellular scenarios. **(b)** Trace from PiTree [34].

(c) Highly variable scenarios. **(d)** Driving trace from NYC [7].

(e) Wifi scenarios. **(f)** Office Wifi trace from [35].

Figure 17: The detailed transport layer results for the three scenarios. (a)(c)(e) presents the normalized avg. delay, and avg. throughput. The center of each ellipse shows the average value of each CCA, while the ellipse indicates the standard deviations from the average values and their covariance. (b)(d)(f) shows the avg. delay (icons), 90% tile delay (end of lines), and avg. throughput of a sample.

values of the RTT and the throughput and their covariance[4]. Floo can not only achieve lower latency and higher throughput, but also obtain a smaller ellipse than other CCAs, which denotes better stability and consistency. In Fig. 17b, Fig. 17d and Fig. 17f we also depict the avg. delay (icons), 90% tail delay (end of lines), and avg. throughput of three typical sample traces. Under all three scenarios, Floo achieves excellent performance both in delay and throughput.

F Analysis of Selection Period.

Here, we investigate the impact of SP value on the performance and overhead of the mechanism. Intuitively, the SP determines the frequency of Floo monitoring network and application states, and selecting CCAs. SP should be consistent with the granularity of the application QoE and should also consider the network fluctuation. To this end, we vary the SP and record its impact on application performance. We

[4]Note that RTT/RTT_Min $\in [1,\infty)$, and Thp/Thp_Max $\in [0,1)$. The ellipse may exceed the range, but the outlier part is actually not sampled.

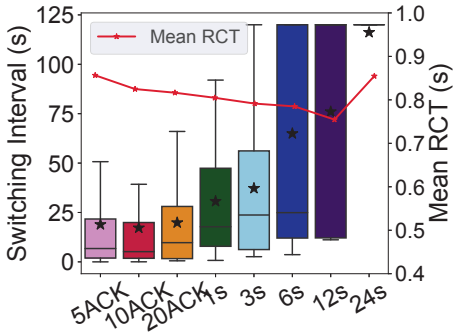


Figure 18: Avg. RCT (red line) and switching intervals distribution across diff. SPs

vary the SP from the fine-grained ACK level to the coarse-grained second-level. To report the performance, we conduct experiments in 60 scenarios and record the RCT and CCA switching interval. Note that if no switching has occurred in one scenario, the switching interval is counted as the duration of that experiment (120s).

Fig. 18 depicts the results. QoE gradually improves with the growth of the SP. The performance of ack-level selection is worse. We find that even for fine-grained ack-level SP, the granularity of CCA switching interval is at the second-level, with a median of about 5 to 10 seconds and an average value of more than 16 seconds. The QoE results are consistent with the frequency of CCA switching. This is because the fine-grained data estimation is susceptible to outliers, and cannot reflect the real path condition and application performance. On the other hand, the frequency of CCA selection is higher than that of request sending, which could lead to meaningless CCA switching. However, long SP, such as 24s, is challenged to capture and react to the instant changes in network conditions and application performance. In 96% of the scenarios, the SP of 24s does not switch during the connection. Therefore, long SP could not achieve good performance. As for the CPU utilization, experiments show that SP has little impact on the overhead. The difference in CPU utilization between different SPs is less than 0.28%. To have a balanced performance and overhead, we set a fixed SP of 12s in this paper, which represents the minimum switching period.

FarReach: Write-back Caching in Programmable Switches

Siyuan Sheng[1], Huancheng Puyang[1], Qun Huang[2], Lu Tang[3], and Patrick P. C. Lee[1]
[1]*The Chinese University of Hong Kong* [2]*Peking University* [3]*Xiamen University*

Abstract

Skewed write-intensive key-value storage workloads are increasingly observed in modern data centers, yet they also incur server overloads due to load imbalance. Programmable switches provide viable solutions for realizing load-balanced caching on the I/O path, and hence implementing write-back caching in programmable switches is a natural direction to absorb frequent writes for high write performance. However, enabling in-switch write-back caching is non-trivial, as it not only is challenged by the strict programming rules and limited stateful memory of programmable switches, but also necessitates reliable protection against data loss due to switch failures. We propose FarReach, a new caching framework that supports fast, available, and reliable in-switch write-back caching. FarReach carefully co-designs both the control and data planes for cache management in programmable switches, so as to achieve high data-plane performance with lightweight control-plane management. Experiments on a Tofino switch testbed show that FarReach achieves a throughput gain of up to 6.6× over a state-of-the-art in-switch caching approach under skewed write-intensive workloads.

1 Introduction

Key-value stores have been widely deployed in modern data centers to manage structured data (in units of *records*) for data-intensive applications, such as social networking [1, 33, 41], web indexing [7], and e-commerce [11]. Practical key-value storage workloads are traditionally read-intensive (e.g., up to a read-to-write ratio of 30:1 at Facebook's Memcached [1]). Recent field studies of production key-value stores show the dominance of *write-intensive workloads*; for example, more than 20% of Twitter's Twemcache clusters have more writes than reads [41], and the AI/machine-learning services at Facebook's RocksDB production have 92.5% of read-modify-writes [6]. Also, write-intensive workloads are *skewed*; for example, 25% of frequently accessed (i.e., *hot*) records dominate in the write workloads at Twitter's Twemcache clusters [41].

Enabling high write performance for key-value stores in data centers is challenging. Write requests issued from a client to a key-value storage server often suffer from long round-trip times (RTTs) due to switch-to-server transmissions and server-side processing. In particular, if the server is overloaded, I/O requests will have long queuing delays or even be dropped. Also, in distributed key-value stores that span multiple servers, a small portion of servers may be bottlenecked by substantial requests for hot records under skewed workloads, thereby leading to load imbalance [17, 21].

Programmable switches [5] offer an opportunity to improve the write performance of key-value stores. By deploying a programmable switch on the I/O path (e.g., as a top-of-rack switch in a rack-based data center), it can inherently intercept the I/O requests for all servers within the rack and provide stateful memory that can be programmed to cache frequently accessed records. For each request issued by a client, the switch can read or write any of its cached records and directly respond to the client, thereby eliminating the long RTT to access the server-side record. It is also proven that load balancing is achievable by keeping only $O(N \log N)$ records, where N is the number of servers [14]. Recent studies have demonstrated the effectiveness of load-balanced in-switch caching [20, 27–29] for high throughput and sub-RTT latencies. However, existing in-switch caching approaches [20, 27–29] target read-intensive workloads and implement *write-through* caching (i.e., write requests update records both in the in-switch cache and the server side). Thus, they do not provide write performance gains compared with no caching under skewed write-intensive workloads.

To address skewed write-intensive workloads, it is desirable to implement in-switch *write-back* caching (i.e., write requests update records in the in-switch cache only without immediately updating the server side) to absorb frequent writes to hot records. However, enabling write-back caching in programmable switches is subject to several challenges. First, in-switch write-back caching raises an issue of synchronizing records in both the in-switch cache and server-side storage. Without proper synchronization, the latest records may become unavailable to clients during cache eviction. Second, since the in-switch cache keeps the latest records under the write-back policy, protecting against data loss in the in-switch cache during switch failures is critical. However, providing fault tolerance guarantees for the in-switch cache is challenged by the limited switch resources (e.g., limited stages with only tens of megabytes for stateful memory) [5, 31]. Finally, the strict pipeline programming model and limited resources in programmable switches necessitate a design of simple but efficient caching mechanisms. While programmable switches can be managed with a software controller to relax switch resource constraints [20, 29], the control-plane interaction between the controller and programmable switches can incur long delays and slow down the data-plane I/O performance. Even worse, the synchronization and fault tolerance issues complicate cache management with extra control-plane overhead, thereby further degrading I/O performance.

In this paper, we propose FarReach, a fast, available, and reliable in-switch write-back caching framework to improve the I/O performance of key-value stores under skewed write-intensive workloads. FarReach exploits a careful co-design of the control and data planes, such that it offloads cache management to a centralized controller in the control plane, while achieving high data-plane performance with lightweight control-plane management. It comprises the following design features: (i) fast cache admission that admits hot records into the in-switch cache without blocking data-plane I/O traffic; (ii) available cache eviction that ensures that the latest records evicted from the in-switch cache remain available to read requests; and (iii) reliable snapshot generation and zero-loss crash recovery for the protection against data loss during switch failures. To the best of our knowledge, this is the first work that specifically addresses the availability and fault tolerance issues of in-switch caching.

We implement FarReach and compile the in-switch cache prototype (written in P4 [4]) into the Tofino switch chipset [39]. We evaluate FarReach with YCSB [42] and synthetic workloads. Compared with NetCache [20], a state-of-the-art in-switch caching framework that targets read-intensive workloads and implements write-through caching, FarReach achieves a throughput gain of up to 6.6× across 128 simulated servers under skewed write-intensive workloads. FarReach also has low access latencies, fast crash recovery, and limited switch resource usage.

We now release the source code of our FarReach prototype at **http://adslab.cse.cuhk.edu.hk/software/farreach**.

2 Background and Motivation

2.1 Programmable Switches

Figure 1 shows the programmable switch architecture, which consists of both a data plane and a control plane. The data plane processes packets with a stringent timing requirement for line-rate forwarding. It contains multiple *ingress* and *egress* pipelines. When a packet arrives at the switch through an ingress port, the packet first enters the corresponding ingress pipeline, which specifies an egress port. Then the *traffic manager*, which interconnects between the ingress and egress pipelines, transfers the packet to the egress pipeline corresponding to the specified egress port. Finally, the packet leaves the switch through the egress port. On the other hand, the control plane contains an operating system within the switch, called the *switch OS*, to manage the forwarding rules of the data plane. The switch OS of each switch interacts with a centralized *controller*, which manages the packet processing of all switches in a network-wide manner.

Each ingress/egress pipeline follows a reconfigurable match tables (RMT) model [5]. When a packet enters an ingress/egress pipeline, it is first processed by a *parser*, which extracts packet header fields into the packet header vector (PHV). The pipeline transfers the PHV across a number of

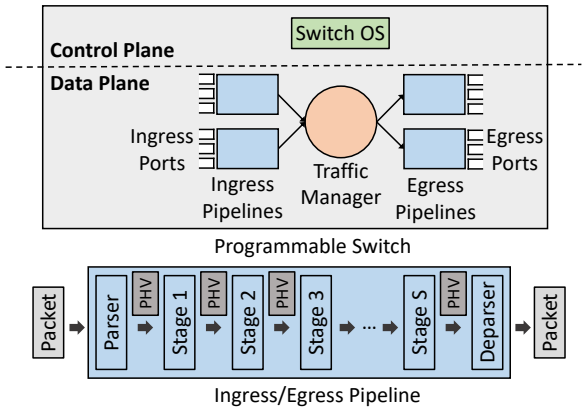


Figure 1: Programmable switch architecture.

stages with multiple *match-action tables* each. Each stage also keeps a limited amount of SRAM, composed of tens of memory blocks, for tracking stateful information that is accessible by the match-action tables. A match-action table can use an ALU to perform arithmetic or logical operations and store the results into the PHV. It matches the fields in the PHV from the previous stage and performs the corresponding action to update the PHV for the next stage, while the match-action rules can be configured by the switch OS. A match-action table can also use a special kind of ALU, called stateful ALU, to store the results into on-chip memory. To fulfill the stringent timing requirement, the memory blocks associated with a stage cannot be accessed from other stages, while the processing of a packet within a stage can only access a limited number of memory blocks associated with the stage and each memory block can only be accessed at most once. After being updated by all stages, the PHV is processed by a *deparser*, which reconstructs the new packet header fields. The header fields are combined with the original payload to form the packet to be forwarded.

2.2 Challenges

Write caching policies can be classified into *write-through* and *write-back*. Write-through synchronously updates the records both in the cache and on the server side; in contrast, write-back (a.k.a. delayed-write) updates only the records in the cache, and later reflects the updates on the server side. Existing in-switch caches [20, 27–29] mainly implement write-through caching. In this work, we focus on write-back caching, as it improves the write performance over write-through caching by delaying server-side updates. However, managing write-back caching is non-trivial, and is subject to three unique challenges in programmable switches.

Performance challenge. Since a programmable switch has a restricted pipeline programming model (i.e., it can only access a limited number of memory blocks) and scarce hardware resources (i.e., it only has a limited number of stages and stateful ALUs) [5], it is necessary to offload switch-level cache management (including cache admission and eviction) to a

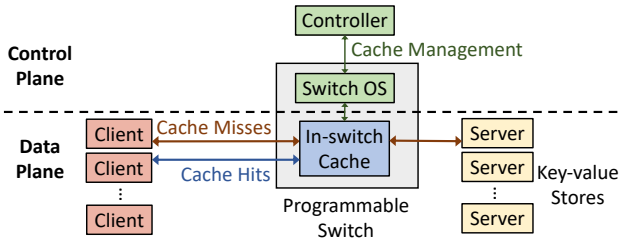


Figure 2: FarReach's architecture.

centralized controller [20, 29], while the switch only updates the cached records in the data plane under the write-back policy. However, due to the high controller-to-switch latency, control-plane processing is much slower than data-plane processing in a programmable switch, thereby bottlenecking the I/O performance.

Availability challenge. Under write-back caching, both cache admission and eviction algorithms need careful coordination between the control and data planes, so as to correctly maintain the latest records in either the in-switch cache or server-side storage; otherwise, the outdated records may be returned to the client. Such an issue does not exist in write-through caching [20, 29], as it always keeps the latest records on the server side. The availability issue is even more challenging in programmable switches, since the controller needs to manage both the cache and server updates, but incurs high overhead. Also, the controller is not on the packet forwarding path and has no view about the traversed packets in the data plane.

Reliability challenge. Under write-back caching, the latest records may only be kept in the in-switch cache and may have their updates to the server-side storage delayed. If the switch crashes, all latest records are lost. Such an issue again does not exist in write-through caching, as the latest records can be persistently kept in server-side storage [20, 29].

3 FarReach Design

3.1 Design Overview

Architecture. FarReach is a fast, available, and reliable in-switch write-back cache architecture for improving the I/O performance and load balancing of server-side key-value stores. Figure 2 shows FarReach's architecture, in which clients are connected via the in-switch cache to multiple servers for key-value storage, while the controller is responsible for cache admission and eviction. Recall that the controller has no view about the data plane (§2.2). Thus, the cache management decisions are triggered by the switches (in the data plane) based on the workload patterns.

Goals. FarReach's core idea is a careful co-design of the control and data planes. Table 1 summarizes our design features. FarReach aims for three design goals:

- *Fast access* (§3.2). FarReach supports non-blocking cache admission for admitting hot records into the in-switch cache, so as to achieve high write performance. It also ensures atomicity in cache admission under the multi-pipeline setting of programmable switches.
- *Availability* (§3.3). FarReach ensures that any latest record that is evicted from the in-switch cache remains available to clients.
- *Reliability* (§3.4). FarReach protects against data loss during switch failures. It uses a crash-consistent snapshot generation algorithm for making snapshots of the in-switch cache state. It also ensures atomicity of snapshot generation in the multi-pipeline setting. It further couples snapshot generation with upstream backup [18] to achieve zero-loss crash recovery.

Table 1: Summary of design features of FarReach.

Design features	Design details
Non-blocking cache admission (§3.2)	FarReach tracks the "outdated" or "latest" state of each cached record to limit conservative reads. It also associates a validity register with each cached record for atomicity.
Available cache eviction (§3.3)	FarReach uses a "to-be-evicted" flag to make each evicted record available. It identifies latest records by sequence numbers and handles packet loss by record embedding.
Crash-consistent snapshot generation (§3.4)	FarReach reports original cached records to the controller. It recirculates writes for atomicity, and exploits client-side record preservation for zero-loss recovery.

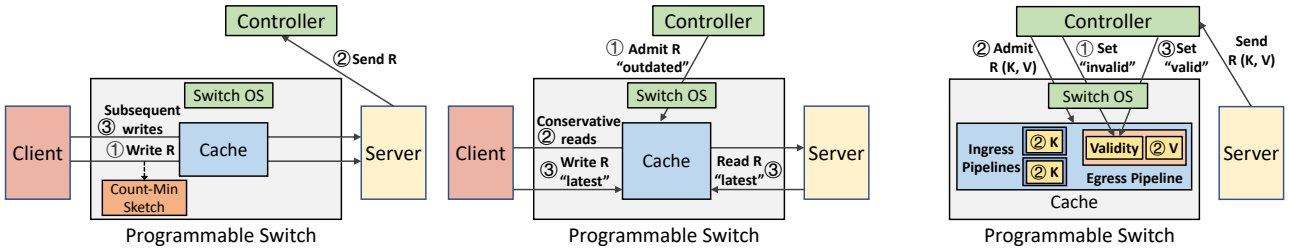
Design assumptions. FarReach currently supports a fixed key length of 16 bytes and a variable value length of up to 128 bytes due to limited switch resources; the same constraint is also assumed in NetCache [20] and DistCache [29]. Thus, FarReach is suitable for workloads dominated by small records (e.g., ZippyDB and UP2X in Facebook's RocksDB production [6]). For large records, FarReach simply relays them between clients and servers without caching.

FarReach currently does not support range queries, since programmable switches cannot feasibly maintain sorted structures with the memory access limitations (§2.1) and servers are unaware of the latest in-switch records under the write-back policy. In this work, we focus on skewed write-intensive workloads without range queries.

FarReach guarantees reliability for switch failures. We assume that the durability of server-side records is addressed by the persistence feature of key-value stores [25, 30, 37].

3.2 Non-blocking Cache Admission

Problem of cache admission. A naïve design of cache admission in programmable switches can introduce *blocking* to write requests. Due to limited switch resources, the controller is responsible for cache management (§2.2). Suppose that the controller is about to admit a new hot record into the in-switch cache. As control-plane processing is slower than data-plane packet forwarding (§2.2), the switch may receive subsequent



(a) Before cache admission (b) After cache admission

Figure 3: Non-blocking cache admission in FarReach. Before admitting a record R, the switch forwards subsequent writes to the server in a non-blocking manner. After admitting R, the switch conservatively forwards reads for R to the server until it receives a new write from the client or a read from the server; it also marks R as "latest".

Figure 4: Atomic validity control in FarReach. For a record R with a key K and a value V, the controller maintains an egress validity register for atomicity of cache admission.

writes for the same key before the record from the first write is admitted to the cache. In this case, such subsequent writes need to be *blocked* until the record is admitted; otherwise, the admitted record may overwrite the newer records from the subsequent writes that arrive earlier at the switch, due to the write-back policy.

Cache admission policy. Before proposing our cache admission design, we first describe the cache admission policy in FarReach. FarReach currently triggers cache admission for the hot records with high access frequencies. It follows the design of NetCache [20] and deploys space-efficient in-switch data structures for frequency tracking, due to the limited in-switch SRAM. Specifically, FarReach maintains a Count-Min Sketch [10] to track the access frequencies of uncached records for cache admission, as well as a counter array to track the access frequencies of cached records for cache eviction (§3.3), within the switch. A Count-Min Sketch is a fixed-size, error-bounded summary data structure composed of multiple rows with a fixed number of counters each. FarReach samples incoming requests for frequency monitoring to reduce processing overhead. For each sampled request to an uncached key, FarReach updates the Count-Min Sketch and estimates the access frequency. If the frequency exceeds a pre-defined threshold, FarReach identifies the key as hot. It triggers the controller to admit the hot record into the in-switch cache, and also tracks the frequency of the cached record in the counter array. To avoid counter overflow, FarReach periodically resets all counters of the Count-Min Sketch and the counter array to zero. Note that we do not claim the novelty of this design.

Our cache admission design. We propose a non-blocking cache admission algorithm for FarReach, as shown in Figure 3. Suppose that a client issues a write request of a record (say, R) to a server. If R is not yet cached and is identified as hot based on the Count-Min Sketch, the switch forwards R to the server (① in Figure 3(a)). The server forwards R to the controller for cache admission (② in Figure 3(a)). Note that a read request issued by a client can also trigger cache admission, except that the server will send the server-side latest record R to the controller (② in Figure 3(a)). Before the controller admits R into the in-switch cache, the switch forwards subsequent writes for the same R's key (i.e., cache misses) to the server without updating the cache (③ in Figure 3(a)). The server directly processes the writes without blocking. Thus, the server now keeps the latest record.

After R is admitted, FarReach temporarily marks the admitted R as "outdated" (① in Figure 3(b)). For any read request to R's key (which is "outdated"), FarReach *conservatively* forwards the read request to the server to obtain the latest record (② in Figure 3(b)).

Conservative reads increase read latencies due to server-side processing. To limit conservative reads, our insight is that all requests and responses must traverse the switch, so FarReach can monitor all traversed requests and responses to mark the "outdated" cached record as "latest" as early as possible. Specifically, FarReach marks the "outdated" record as "latest" (③ in Figure 3(b)) if it sees: (i) a write request from a client for the same key (which carries the latest record), or (ii) a read response from the server for the same key (which carries the latest record while the cached record remains outdated). When a cached record is marked as "latest", it can be directly updated by subsequent writes based on the write-back policy. Under skewed write-intensive workloads, an "outdated" cached record can soon be marked as "latest" by a subsequent write for the same key, so conservative reads are limited.

Recall that a server in FarReach is responsible for sending a record to the controller for cache admission (i.e., ② in Figure 3(a)). Thus, it can determine whether any record of the same key has been sent to the controller and avoid sending duplicate records of the same key, thereby keeping limited control-plane bandwidth usage (e.g., up to 1.41 MiB/s; see §5.4). This is in contrast to NetCache [20], in which a switch sends records to the controller for cache admission and needs an in-switch Bloom Filter [3] to avoid duplicate submission; FarReach removes the need of maintaining an in-switch Bloom Filter and hence saves switch resource usage for implementing in-switch write-back caching.

Atomic validity control. FarReach stores the keys and values of records in the ingress and egress pipelines, respectively, to accommodate the limited number of stages of a single pipeline. However, it is critical but non-trivial to provide atomicity for

cache admission under the multi-pipeline setting. Specifically, a switch can only provide atomicity within a single pipeline rather than multiple pipelines, yet the requests for the same key can arrive from different ingress pipelines. Without the atomicity of cache admission, the write requests to the same key arriving from different ingress pipelines may have inconsistent views on the key: cached or uncached. For the former, the cached record is updated directly by the write-back policy; for the latter, the requests are forwarded to the server based on our non-blocking cache admission design. Thus, the key may be updated with an inconsistent value.

Our insight is that although the requests for the same key can enter a switch from different ingress pipelines, FarReach can forward them to the same egress pipeline corresponding to the same server. Note that such forwarding does not incur cross-pipeline imbalance, as the bottleneck lies in server-side storage (including both CPU processing and disk I/O) instead of line-rate switches. The server-side bottleneck is shown in our evaluation, where the system throughput is up to 12.1 MB/s under 128 simulated servers (§5.2), significantly lower than the maximum throughput 3.2 Tbps of a two-pipeline Tofino switch [39]. Thus, FarReach can provide atomicity for each record being admitted, with the aid of the single egress pipeline that is connected to the corresponding server, while incurring limited performance degradation.

We propose *atomic validity control* for cache admission in FarReach (Figure 4). Specifically, programmable switches provide atomic primitives for each register within a single pipeline. FarReach introduces a *validity register* for each cached key in an egress pipeline. Before admitting a record R with key K and value V sent by a server, FarReach first sets the validity register for R as "invalid" (① in Figure 4). It then admits, via the switch OS, V into the egress pipeline and K into all ingress pipelines (the latter is to ensure consistency across all ingress pipelines) (② in Figure 4). Finally, it changes the validity register to "valid" (③ in Figure 4). Based on the validity register, FarReach treats a record as a cache hit only if the key is cached in an ingress pipeline and the validity register is "valid" in the single egress pipeline; or as a cache miss otherwise. Thus, if a key has not been admitted into all ingress pipelines, its record is treated as a cache miss as its validity register remains "invalid".

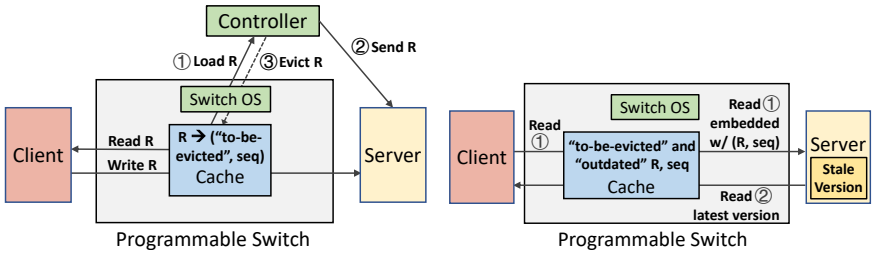
3.3 Available Cache Eviction

Problem of cache eviction. If the in-switch cache is full for cache admission, FarReach selects a cached record to evict, by sampling multiple cached records and selecting the one with the least access frequency from the counter array (§3.2). It then triggers the controller to perform cache eviction on the selected record. Under the write-back policy, the evicted record may also be the latest record and has not yet been updated in the server. It is critical to keep any latest record to be evicted available during cache eviction. To achieve this goal, the controller needs to synchronize the views of both the switch and the server on the evicted record during cache eviction, especially when there also exist read/write requests for the evicted record. However, the controller is constrained by slow control-plane processing, which leads to high synchronization overhead.

Our cache eviction design. We propose a cache eviction algorithm for FarReach that ensures availability, whose workflow is shown in Figure 5(a). Our idea is to associate additional metadata with each cached record in the in-switch cache, so as to maintain the availability of any evicted record, while incurring limited synchronization overhead to the controller. Specifically, when a cached record (say, R) is to be evicted, the controller first marks R as "to-be-evicted" and loads R from the in-switch cache (① in Figure 5(a)). It then sends R to a server for storage (② in Figure 5(a)). If there is any write request to the "to-be-evicted" R, FarReach simply forwards the write request to the server (instead of updating the record in the cache under the write-back policy) and marks the evicted record as "outdated". If there is any read request to the "to-be-evicted" R and R is "latest" (marked in cache admission (§3.2)), the cache returns R to the client; otherwise, if R is "outdated" (i.e., it has been updated), FarReach forwards the read request of R to the server, which holds the latest record. Thus, we ensure that any evicted cached record that is also the latest record remains available. After the server has stored the latest "to-be-evicted" cached record, the controller acknowledges the cache to actually evict the "to-be-evicted" R (③ in Figure 5(a)). Note that all writes to the "to-be-evicted" R must be forwarded to the server no matter with the view of cached or uncached, so FarReach does not have any atomicity issue when evicting R in the multi-pipeline setting.

Identifying latest records. One subtlety is that a server may receive the request of storing a record from two possible paths: (i) the eviction of a record from the cache and (ii) the write request of the record issued by a client. It is critical to differentiate the latest version of a record that is finally stored in the server. To resolve this issue, recall that FarReach forwards the write requests of the same record to the same egress pipeline corresponding to the server (§3.2). As programmable switches can provide atomicity and serialize packets in a single pipeline (§3.2), FarReach associates a *sequence number* with each cached record atomically. It increments the sequence number for each write request of the key in the egress pipeline based on the serialized order of accessing the cache, and embeds the sequence number into the write request. When the server receives a request of storing a record, it overwrites the existing record only if the received record has a higher sequence number than the existing record; otherwise, the received record will be discarded.

Handling packet loss. Packet loss in switch-to-server transmissions can break the availability of cache eviction. To elaborate, recall that an in-switch record can be the latest version under the write-back policy. During cache eviction, FarReach



(a) Cache eviction workflow (b) Record embedding

Figure 5: Available cache eviction in FarReach. For a record *R* to be evicted, it is marked as "to-be-evicted" and is made available to the client's read if it is also the latest record. To handle switch-server packet loss, if *R* is "outdated", the switch embeds the "outdated" evicted record into any read and forwards the read to the server. The server compares the received read with the server-side version and keeps the latest version.

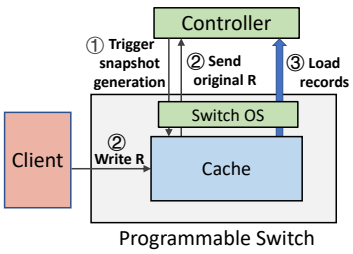


Figure 6: Crash-consistent snapshot generation in FarReach. If the switch receives the first write to a cached record *R* during snapshot generation, it forwards the original *R* to the controller before *R* is updated.

forwards the write request to a "to-be-evicted" record to the server and marks the evicted record as "outdated". If the write request is lost during its transmission (e.g., due to server-side congestion or packet corruption), the server is not updated with the latest version, but still keeps the stale version. As the in-switch cache marks the evicted record as "outdated", it forwards all subsequent reads to the server and receives the stale version. Note that such an issue does not exist in cache admission, as a write request updates either the server (before the record is admitted to the in-switch cache) or the in-switch cache (after the record is admitted to the in-switch cache), instead of changing both of them.

To maintain availability under packet loss, FarReach employs *record embedding* during cache eviction, as shown in Figure 5(b). Our insight is that even though an evicted record is marked as "outdated" during cache eviction, it can still be the latest version that can be used for serving read requests. Specifically, before forwarding a read to the server, the in-switch cache embeds the "outdated" evicted record (if such a record exists) into the read; the embedded record includes the value and sequence number assigned by the switch (① in Figure 5(b)). FarReach ensures that the latest version is available to any client-issued read by comparing the embedded record with the server-side version (② in Figure 5(b)): if the sequence number embedded into a read request is larger than that stored in the server (i.e., the embedded record is the latest version), FarReach directly returns the embedded record to the client; otherwise, FarReach returns the record stored in the server (which is the latest version) to the client.

3.4 Crash-consistent Snapshot Generation

We now address the reliability challenge (§2.2) through a consistency model that incurs zero data loss after switch failures. At a high level, FarReach periodically generates snapshots to protect against data loss of in-switch cached records. It also lets each client preserve the cached records generated after the latest snapshot for recovery. Note that the uncached records are protected by server-side persistent key-value stores (§3.1).

Problem of snapshot generation. Since the in-switch cache keeps the latest records under the write-back policy, we need to protect against data loss in switch failures. We propose to generate a snapshot for all cached records in the in-switch cache at regular time points (called *snapshot points*), so that the switch can restore from the latest snapshot when recovering from a switch failure. However, the design of such snapshot generation is non-trivial. Since programmable switches have limited stages for cache backup and limited on-chip memory for snapshot storage, they need to offload all cached records to the controller. Note that the snapshot overhead is limited for the controller, as the controller only needs to store the latest snapshot for crash recovery (e.g., 1.5 MB for 10K records with 16-byte keys and 128-byte values). When the cached records are loaded to the controller during snapshot generation, some cached records may be updated under the write-back policy. The final snapshot will become inconsistent with the in-switch cache state at the snapshot point. Blocking cache updates during snapshot generation can avoid such inconsistencies, yet it also degrades the I/O performance.

Our snapshot generation algorithm. We propose a two-phase snapshot generation algorithm for FarReach to maintain crash consistency in snapshot generation, without blocking cache updates. Our insight is that whenever FarReach receives the first write request to a cached record during snapshot generation, it can send the original cached record (i.e., after the snapshot point but before the first write) to the controller. This allows the controller to keep the backups for all original cached records that are to be overwritten. At the end of snapshot generation, the controller replaces the overwritten cached records by their backups of the original cached records, so that the snapshot is crash-consistent with the in-switch cache state at the snapshot point. Under the skewed write-intensive workloads where most writes are issued to a small fraction of hot records, FarReach only needs to send limited original cached records to the controller (without the need to send the cached record for the subsequent writes after the first write). Thus, the bandwidth overhead in the controller is limited.

Based on the insight, FarReach generates a crash-consistent snapshot in a two-phase manner (i.e., triggering snapshot gen-

eration and making a consistent snapshot) at each snapshot point, as shown in Figure 6. In the first phase, the controller notifies the in-switch cache to trigger snapshot generation (① in Figure 6). The cache monitors each traversed write request to identify whether the write request is the first write to a cached record during snapshot generation. If so, the cache sends the original cached record to the controller (② in Figure 6). In the second phase, the controller loads all cached records from the cache for snapshot generation (③ in Figure 6). Note that if a cached record has been loaded to the controller and later receives the first write, the cache no longer needs to send the original cached record, which has already been loaded. Once the controller loads all cached records, it notifies the cache about the completion of snapshot generation (i.e., the cache no longer needs to monitor the writes to cached records), reverts any overwritten cached record with the original one, and finally obtains a crash-consistent snapshot.

FarReach carefully updates the snapshot to address two corner cases. If a new record is first admitted to the cache after the snapshot point, the controller will not include the record into the snapshot. If a cached record is evicted after the snapshot point, the controller saves the evicted record during cache eviction (§3.3), and replaces the updated record with the evicted record in the snapshot after the second phase of snapshot generation.

Atomic triggering of snapshot generation. As the write requests of a record can arrive from multiple ingress pipelines, FarReach needs to trigger snapshot generation in multiple pipelines at the same time; otherwise, the ingress pipelines may set a snapshot point at different times and generate inconsistent snapshots. We propose a coordination mechanism to support simultaneous snapshot generation in multiple ingress pipelines. Specifically, FarReach selects one of the ingress pipelines, and recirculates all write requests from other ingress pipelines to the selected ingress pipeline; in other words, all write requests are processed as if they arrive at a single ingress pipeline. The controller first notifies the selected ingress pipeline to trigger snapshot generation, such that the selected ingress pipeline notifies the egress pipelines to send any original cached record that receives the first write to the controller. It then notifies the remaining ingress pipelines to trigger snapshot generation. After all ingress pipelines trigger snapshot generation, FarReach disables the recirculation, and now the controller can perform snapshot generation with all ingress pipelines in parallel. Thus, we ensure that snapshot generation is applied to all ingress pipelines at the same snapshot point. Note that the recirculation overhead is limited, due to the short duration for notifying all ingress pipelines to trigger snapshot generation (e.g., $\approx$6 ms from our evaluation).

Zero-loss crash recovery. Our snapshot generation only guarantees crash consistency for switch failures, but the cached records that are newly added or updated after the latest snapshot point remain unprotected and can be lost during a switch failure. Unfortunately, switches do not have external storage for keeping cached records reliably. To achieve zero-loss crash recovery, we propose *client-side record preservation* based on the idea of upstream backup [18] in stream processing, by keeping the copies of cached records after the latest snapshot point on the client side. Specifically, after a client sends a write request of a cached key and receives the response from the in-switch cache, it keeps locally the value and sequence number assigned by the in-switch cache (§3.3) for the cached key. After the completion of snapshot generation at each snapshot point, the controller notifies each client with the cached keys and the corresponding sequence numbers at the snapshot point. Each client then releases its preserved records whose sequence numbers are no larger than those notified by the controller. Since the in-switch cache only keeps a limited number of hot records, FarReach incurs low client-side overhead for record preservation.

FarReach exploits a *replay-based approach* to achieve zero-loss crash recovery after a switch failure. It first replays the write requests of the latest cached records to update the servers for persistent storage. Specifically, FarReach collects both the latest in-switch snapshot (from the controller) and the client-side preserved records (from all clients), and selects the record with the largest sequence number for each cached key. If the sequence number of each selected record is larger than that stored in a server, FarReach replays the write request to store the selected record in the server for persistent storage. After all latest records are persisted, the clients can then release their preserved records.

FarReach next recovers the in-switch cache, by replaying the cache admission for each record of the latest in-switch snapshot, and marks each cached record as "outdated". The "outdated" records of the in-switch cache are expected to be quickly marked as "latest" under skewed write-intensive workloads (§3.2). Note that we do not simply start with an empty in-switch cache from scratch, as it incurs large overhead to admit all records through the controller.

Client crashes. One limitation of FarReach is that data loss can occur if both a client and the switch crash simultaneously. If any client crashes before replay-based recovery, the cached records preserved by the client, which are not yet protected by the latest in-switch snapshot, will be lost after a switch failure. We can reduce the snapshot period to a smaller window for less vulnerability, at the expense of incurring larger snapshot generation overhead. Nevertheless, the snapshot generation overhead remains still limited (e.g., up to 1.41 MiB/s of control-plane bandwidth when the snapshot period is 2.5 s; see §5.4). We leave how to completely prevent data loss from client crashes as future work.

3.5 Discussion

Novelty. While FarReach borrows ideas from NetCache [20] (e.g., cache admission based on a Count-Min Sketch), it has other novel design elements: (i) non-blocking cache admission for fast access, with atomic validity control to address

the atomicity issue (§3.2); (ii) available cache eviction for the availability of records, with record embedding to handle packet loss (§3.3); and (iii) crash-consistent snapshot generation with zero-loss recovery (§3.4). Note that the last two elements are tailored for write-back caching and are not found in NetCache for write-through caching.

Trade-offs. FarReach makes two trade-offs in its design. First, FarReach trades extra switch resources for in-switch caching for higher key-value storage performance under skewed write-intensive workloads, yet we show that the extra switch resource usage of FarReach for supporting the write-back policy is similar to that of NetCache (§5.5). Second, FarReach trades extra client-side storage capacity for zero-loss recovery. Nevertheless, since the clients only keep the copies of cached records after the latest in-switch snapshot, the client-side storage overhead is limited (e.g., 1.5 MB for 10K cached records with 16-byte keys and 128-byte values).

Future work. We pose two open issues as future work. First, in addition to reducing vulnerability window from client crashes (§3.4), FarReach should collect the preserved records from all clients during crash recovery, and it may limit scalability as the number of clients increases. One possible solution is to extend FarReach with multiple switches and partition clients among them, such that FarReach can collect the preserved records through multiple switches in parallel. Second, FarReach offloads cache management (including cache admission and eviction) to a centralized controller, which may be overwhelmed by extensive cache admission and eviction decisions. One possible solution is to rate-limit admission and eviction operations to avoid overloading the control plane.

4 Implementation

We prototyped FarReach with both the control and data planes. The control plane includes the switch OS and the controller, while the data plane includes multiple clients and servers as well as the in-switch cache. All communications among different components are based on UDP with a timeout-and-retry mechanism for low-latency yet reliable transmissions.

4.1 Control Plane

We implement both the switch OS and the controller in C++, with 2.2K and 1K LoC, respectively, and compile the programs by `g++` (v5.4.0) with the `-O3` optimization. The switch OS provides interfaces for: (i) cache admission/eviction by configuring match-action tables and setting registers, and (ii) snapshot generation by loading in-switch records and sending original cached records. The controller manages the in-switch cache through the interfaces provided by the switch OS and coordinates snapshot generation by communicating with the switch OS and all key-value storage servers.

4.2 Data Plane

Client implementation. We evaluate our prototype with the YCSB benchmark [42] (§5), which is written in Java. We implement a client application in Java that supports YCSB, with the common key-value storage interfaces including `get`, `put`, and `delete` to access records stored in both the in-switch cache and key-value storage servers. The client application also provides a shim layer to manage client-side record preservation for zero-loss recovery under switch failures (§3.4).

Server implementation. We deploy RocksDB (v6.22.1) [37] in each server; RocksDB is a log-structured merge-tree (LSM-tree) persistent key-value store [34] that is suitable for write-intensive workloads. To support multiple servers, we distribute records across servers using consistent hashing [22].

In-switch cache. We implement the in-switch cache in P4 [4] and compile it into the Tofino switch chipset [39]. The cache implementation includes both ingress and egress pipelines. In each ingress/egress pipeline, the Tofino switch provides 12 stages for pipeline programming. Each stage has 4 stateful ALUs to support at most 4 register arrays, and each register can store 4 bytes of data.

In each ingress pipeline, we deploy multiple match-action tables for egress processing. We implement a match-action table for cache lookup, which matches the key (currently of size 16 bytes) in the packet header to obtain the record location in the egress pipeline. We also deploy a match-action table to trigger snapshot generation, such that each egress pipeline can send the original cached records to the controller (§3.4). As the Tofino switch currently does not support cross-pipeline recirculation, we connect the selected ingress pipeline with each of the other ingress pipelines with a physical wire, so as to recirculate the write requests from the other ingress pipelines to the selected ingress pipeline during snapshot generation in the multi-pipeline setting (§3.4). Furthermore, we pre-compute the hash results for the Count-Min Sketch in the ingress pipeline and send them to the egress pipeline by each packet header, so as to save the stages in the egress pipelines.

In each egress pipeline, we store the statistics, metadata, and cached values. In the first stage, we deploy a Count-Min Sketch and configure it with 4 rows as suggested in [20]. Each row corresponds to a register array with 64K registers. We use part of the second stage to maintain a counter array (as a register array) to track the access frequencies of cached records (§3.2). To support write-back caching and snapshot generation, we use the remaining part of the second stage and the third and fourth stages to maintain the required metadata. We use the remaining 8 (out of 12) stages to provide 32 register arrays of 4-byte registers in total for supporting a value size of up to 128 bytes.

We need to address two subtle issues in the egress pipeline implementation. First, to support write-back caching, the in-switch cache needs to directly respond to a write request with a cache hit. However, the Tofino switch cannot directly change the egress port in the egress pipeline. Thus, we drop the original write request and send a response to the client by cloning. Second, to assign a sequence number for each write request, we can maintain a global counter to track the latest

sequence number, but this easily leads to overflow. Instead, we use multiple global counters to reduce the likelihood of overflow. Specifically, we maintain a register array with 32K registers. We map the write requests of different keys into different registers by hashing, and then increment the hashed register to assign a sequence number for each write request.

5 Evaluation

5.1 Methodology

Testbed. We conduct evaluation on a testbed composed of a 3.2 Tbps two-pipeline Tofino switch [39] and four physical machines. Each machine has four 12-core CPUs (Intel E5-2650 v4), 64 GiB DRAM, and 2 TB hard disk (HGST Ultrastar), and is connected with the switch by a 40 Gbps NIC (Intel XL710). We use two physical machines as clients and another two as key-value storage servers. We connect one client and one server with one pipeline of the switch, and connect the other machines with another pipeline.

Setup. We evaluate FarReach using both YCSB [42] and synthetic workloads (see §5.2 and §5.3, respectively). Since our testbed comprises only two servers, we exploit server rotation [20] to simulate a much larger number of servers. Specifically, let N be the number of simulated servers. Given a workload, we issue the requests to N logical partitions via consistent hashing [22] (§4). We find the partition (called the *bottleneck partition*) that receives the most requests among all N partitions. We run each experiment over N iterations. In the first iteration, we deploy the bottleneck partition in a physical server and send sufficient requests to saturate the bottleneck to measure its performance. In the subsequent $N-1$ iterations, we deploy the bottleneck partition in a physical server and each of the $N-1$ non-bottleneck partitions in another physical server, and measure the performance of the non-bottleneck partition. After N iterations, we add all per-partition performance to obtain the aggregate performance. By default, we simulate 16 servers, and increase the number of simulated servers for scalability evaluation (Exp#3). Note that server rotation is only applied to static workloads without the dynamics in key popularity, and we also study the impact of dynamic workloads (Exp#7).

We compare FarReach against two baselines: NoCache (i.e., no in-switch caching) and NetCache [20] (i.e., the in-switch cache that implements write-through caching). Before each experiment, we pre-load 100M records, each of which contains a 16-byte key and 128-byte value, into each server that is initially empty. For FarReach and NetCache, we fix the in-switch cache size as 10,000 records and pre-load the hottest records into the cache. We also set the sampling rate as 0.5 and the pre-defined threshold as 20 requests for the Count-Min Sketch. For FarReach, we set the snapshot period as 10 s by default. We run all experiments with 5 times, and plot the average results with the 95% confidence levels based on the Student's t-distribution.

Summary of results. We summarize the results as follows:

- Under YCSB workloads, FarReach increases the I/O throughput by up to 91% and 84% (for workload A with 50% reads and 50% writes) compared with NoCache and NetCache, respectively (Exp#1). FarReach also achieves sub-RTT latency with up to 72% reduction of average latency (Exp#2) and scales to an increasing number of servers with up to 6.6× throughput gain (Exp#3).
- Under synthetic workloads, FarReach achieves higher throughput gains over NoCache and NetCache for more write-intensive and more skewed workloads (Exp#4 and Exp#5, respectively), while maintaining similar throughput gains for different value sizes and dynamic workloads (Exp#6 and Exp#7, respectively).
- FarReach's snapshot generation incurs limited overhead on throughput and control-plane bandwidth (Exp#8). Its recovery time is within 2.35 s (Exp#9).
- FarReach incurs similar switch resource overhead as NetCache (Exp#10).

5.2 Performance under YCSB Workloads

(Exp#1) Throughput analysis. We first evaluate the end-to-end throughput using YCSB workloads, namely Load (inserting records), A (50% reads, 50% writes), B (95% reads, 5% writes), C (100% reads), D (95% reads, 5% writes), and F (50% reads, 50% read-modify-writes); we do not consider range queries (i.e., Workload E) due to switch limitations (§3.1). For each workload, we generate requests with 16-byte keys and 128-byte values. The Load workload follows the uniform distribution, workload D follows the read-latest distribution, and workloads A, B, C, and F are skewed and follow the Zipf distribution with the Zipfian constant 0.99 (default in YCSB). We verify that under NoCache, the load throughput to a RocksDB instance can reach 0.06 MOPS, which is consistent with prior findings [2, 35].

Figure 7 shows that FarReach increases the throughput of NoCache by 91%, 55%, 85%, and 72% in the four skewed workloads A, B, C, and F, respectively, by reducing and balancing the server-side load with in-switch write-back caching. FarReach also increases the throughput of NetCache by 84%, 20%, and 61% in workloads A, B, and F, and achieves similar throughput as NetCache in workload C (which is read-intensive). In NetCache, the writes of the cached keys keep invalidating the in-switch write-through cache, especially in write-intensive workloads A and F, and hence limit the throughput of NetCache. NetCache only achieves high throughput in read-intensive workloads B and C. In the non-skewed workloads Load and D, both FarReach and NetCache have similar throughput as NoCache due to limited cache hits.

(Exp#2) Latency analysis. We next evaluate the request latencies. We focus on YCSB workload A, which is skewed and most write-intensive. In particular, we examine the trade-off between the latency and target throughput (i.e., configured by a given sending rate) as in prior studies [8, 12, 20]. We

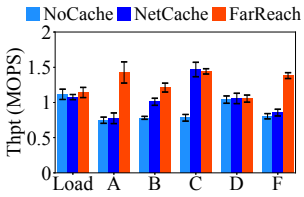


Figure 7: (Exp#1) Throughput analysis.

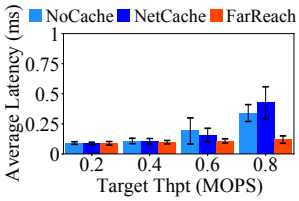


Figure 8: (Exp#2) Latency analysis.

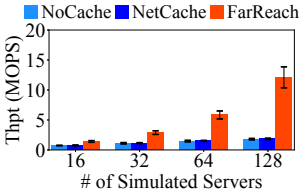


Figure 9: (Exp#3) Scalability analysis.

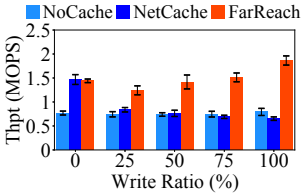


Figure 10: (Exp#4) Impact of write ratio.

only show the average latency results, while the results of other latency statistics (e.g., medium and 95th-percentile) are similar and hence omitted for brevity.

Figure 8 shows that all schemes have small average latencies under low target throughput, as the servers do not have heavy loads and can quickly process requests. FarReach reduces the average latencies of NoCache and NetCache by 65% and 72% when the target throughput is 0.8 MOPS, respectively. For high target throughput, both NoCache and NetCache are bottlenecked by an overloaded server and hence incur large queuing delays. NetCache has a larger latency than NoCache, as NetCache needs extra server-side overhead to update the in-switch write-through cache for the write requests. FarReach effectively reduces and balances the server-side load and hence achieves a small latency. Note that NoCache and NetCache show larger confidence intervals than FarReach, especially for high target throughput. The reason is that the server-side queuing latency can vary significantly for highly overloaded bottleneck server across different runs, while FarReach maintains a low latency due to load balancing.

(Exp#3) Scalability analysis. We evaluate the scalability of different schemes by varying the number of simulated servers. We focus on YCSB workload A. Figure 9 shows that the throughput gains of FarReach are 1.9×, 2.5×, 3.9×, and 6.6× those of NoCache and NetCache (both of which have very similar throughput) under 16, 32, 64, and 128 servers, respectively. As the number of simulated servers increases, the throughput of FarReach also increases due to load balancing across all servers, while the throughput of both NoCache and NetCache is limited by the overloaded servers due to load imbalance. Our results show that FarReach scales to a large number of servers under skewed write-intensive workloads.

5.3 Performance under Synthetic Workloads

We generate different synthetic workloads with YCSB for varying write ratios (over all reads and writes), key distributions, value sizes, key popularities. By default, we generate requests with 16-byte keys and 128-byte values, where the keys follow the Zipf distribution with the Zipfian constant 0.99, and set the write ratio as 100% (i.e., write-only requests).

(Exp#4) Impact of write ratio. We first vary the write ratio of the synthetic workload to evaluate the throughput of different schemes. Figure 10 shows that FarReach increases the throughput of NoCache by 67-135% for different write ratios,

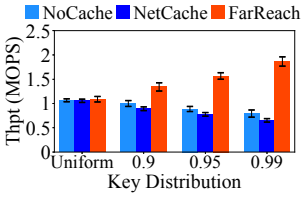


Figure 11: (Exp#5) Impact of key distribution.

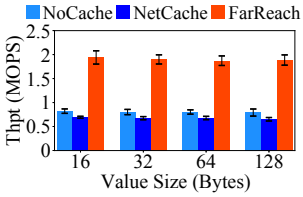


Figure 12: (Exp#6) Impact of value size.

due to load balancing in FarReach. FarReach achieves similar throughput as NetCache when the write ratio is zero (i.e., read-only requests), while increasing the throughput of NetCache by 48-186% as the write ratio ranges from 25% to 100%. The throughput gain of FarReach over NoCache and NetCache is the highest when the write ratio is 100% through in-switch write-back caching. Note that NetCache has slightly smaller throughput than NoCache, especially under the write ratio of 100%, due to the extra server-side overhead to maintain cache coherence for write requests.

(Exp#5) Impact of key distribution. We next consider synthetic workloads under the uniform key distribution as well as the Zipfian key distributions with different Zipf constants. Figure 11 shows that all schemes achieve similar throughput of ≈1 MOPS under the uniform key distribution, as most requests are from the uncached keys and will be processed by the servers, so FarReach cannot benefit from in-switch write-back caching. For the skewed workloads, FarReach increases the throughput of NoCache by 34-135% and that of NetCache by 50-186%. The throughput gain of FarReach is higher when the workload is more skewed (i.e., a larger Zipfian constant), as NoCache and NetCache becomes more imbalanced.

(Exp#6) Impact of value size. We further vary the value size of the synthetic workload from 16 bytes to 128 bytes (while the key size remains 16 bytes); note that the number of records that can be cached in both NetCache and FarReach (i.e., 10,000 records) remains unchanged. Figure 12 shows that the throughput gains of FarReach over NoCache and NetCache remain almost the same at 2.33× across different value sizes, as the caching behavior of FarReach mainly depends on the key distribution.

We also evaluate all schemes when the value size increases to 256 bytes (i.e., exceeding the maximum value size of 128 bytes). All schemes achieve similar throughput of ≈0.7 MOPS (not shown in a figure), as both NetCache and

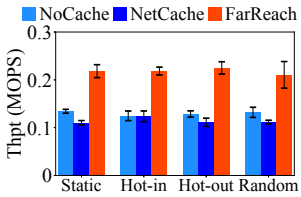


Figure 13: (Exp#7) Impact of key popularity changes.

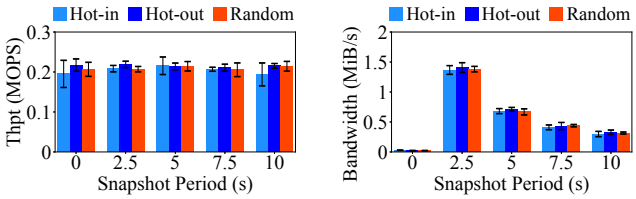


Figure 14: (Exp#8) Performance of snapshot generation, in terms of throughput (left) and control-plane bandwidth (right).

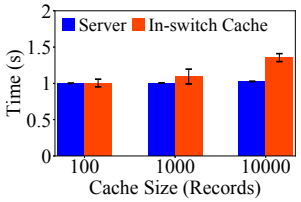


Figure 15: (Exp#9) Crash recovery time.

FarReach directly forward all records to the servers and have similar behavior as NoCache.

(Exp#7) Impact of key popularity changes. Finally, we consider dynamic key popularity patterns, in which the access frequency of a specific key may change over time, while the previous experiments thus far focus on a static key popularity pattern. We consider three dynamic patterns as used in prior work [20, 27]: (i) *hot-in*, which periodically moves the 200 coldest keys to the highest key popularity ranks and decreases the ranks of other keys accordingly; (ii) *hot-out*, which periodically moves the 200 hottest keys to the lowest key popularity ranks and increases the ranks of other keys accordingly; and (iii) *random*, which randomly replaces 200 keys of the top 10,000 hottest keys with coldest keys. As the dynamic patterns will trigger cache management decisions and hence change the system state, we cannot simulate multiple servers by server rotations as in prior experiments. Instead, we evaluate the performance on the two physical servers, each of which runs a RocksDB instance. For each dynamic pattern, we run each scheme for 70 s, and change the key popularity ranks based on each dynamic pattern every 10 s. We measure the instantaneous throughput every 1 s, and evaluate the average throughput over the entire 70 s.

Figure 13 shows that FarReach increases the average throughput of NoCache and NetCache by at least 59% under different dynamic patterns. We also run each scheme for 70 s without any key popularity change (i.e, static), and FarReach has similar throughput gains as in the dynamic patterns. The reason is that FarReach quickly reacts to the key popularity changes (typically within 1 s from our measurement), so it maintains the cache hit rate and hence the average throughput. Note that the throughput is smaller than that in prior experiments as we now use fewer servers, yet our emphasis here is to examine the adaptiveness of FarReach to key popularity changes rather than the absolute performance.

5.4 Snapshot Generation and Crash Recovery

(Exp#8) Performance of snapshot generation. We vary the period of snapshot generation to evaluate the throughput and control-plane bandwidth of FarReach on synthetic workloads. We focus on the results under dynamic patterns, in which the bandwidth costs of both snapshot generation and cache management are included, while we observe similar results under the static pattern and they are omitted for brevity.

Figure 14 shows both the throughput and control-plane bandwidth of FarReach versus the snapshot period; note that if the snapshot period is zero, it means that snapshot generation is disabled. FarReach keeps its throughput at about 0.2 MOPS for various snapshot periods under different dynamic patterns, implying that snapshot generation has a limited impact on throughput.

When snapshot generation is disabled (i.e., the snapshot period is zero), FarReach only incurs about 0.03 MiB/s of control-plane bandwidth, since it only triggers cache management decisions for new hot records and avoids sending duplicate records to the controller (§3.2). When snapshot generation is enabled and as the snapshot period increases from 2.5 s to 10 s, the control-plane bandwidth of FarReach decreases from 1.41 MiB/s to 0.33 MiB/s. Note that the control-plane bandwidth of FarReach is far smaller than the maximum bandwidth of the controller (i.e., 40 Gbps).

(Exp#9) Crash recovery time. We evaluate the crash recovery time of FarReach under a switch failure for various in-switch cache sizes. Specifically, for a given in-switch cache size, we first run the synthetic workload under the static pattern with 16 servers simulated by server rotations. We manually kill the in-switch cache and the switch OS to mimic a switch failure. We then trigger zero-loss crash recovery (§3.4), which applies a replay-based approach to update the servers and recover the in-switch cache. For multiple servers, we take the average time of updating a server as the server-side recovery time.

Figure 15 shows that the time of updating a server in FarReach stays at about 1 s as the cache size increases, as FarReach only replays a limited number of writes partitioned in each server, while taking the majority of time to collect client-side preserved records and control-plane in-switch snapshot. The time to recover the in-switch cache in FarReach increases from 1 s to 1.35 s as the cache size increases from 100 records to 10,000 records, as FarReach needs to admit more records from the latest snapshot under a larger cache size. Overall, the crash recovery time is within 2.35 s for various in-switch cache sizes.

5.5 Switch Deployment

(Exp#10) Switch resource usage. We compile the three schemes into the same Tofino switch chipset [39] to evaluate the switch resource usage. We focus on the following

Table 2: (Exp#10) Switch resource usage (percentages in brackets are fractions of total resource usage).

	SRAM (KiB)	# stages	# actions	# ALUs	PHV size (bytes)
NoCache	320 (2.08%)	4 (33.33%)	6 (nil)	0 (0%)	134 (17.45%)
NetCache	8800 (57.29%)	12 (100%)	69 (nil)	45 (93.75%)	528 (68.75%)
FarReach	8992 (58.54%)	12 (100%)	70 (nil)	45 (93.75%)	499 (64.97%)

metrics: SRAM consumption (with up to 768 KiB for stateful information and 512 KiB for match-action tables per stage), the numbers of stages (12 stages in total), actions, and ALUs (at most 4 stateful ALUs per stage) for in-switch computation, and the packet header vector (PHV) size (768 bytes in total) for cross-stage communication.

Table 2 shows the results. NoCache has the smallest hardware resource usage, as it only needs to support basic network functions (e.g., L2/L3 forwarding). NetCache and FarReach have similar switch resource usage, as both of them deploy an in-switch cache that consumes SRAM to track stateful information (e.g., key-value records and cache metadata). Also, both schemes maintain SRAM-based match-action tables, exploit the stages, actions, and ALUs to perform in-switch computations (e.g., cache lookups and updates), and use the PHV size to transmit each request across different stages.

6 Related Work

In-switch caching and storage management. Several in-switch caching designs have been proposed for high-performance storage. SwitchKV [27] caches hot keys in a software switch, which forwards the reads of cached keys to the in-memory cache nodes, instead of servers, for accessing the values. IncBricks [28] caches records in general-purpose network accelerators and implements packet parsing in programmable switches to serve the reads of cached keys. NetCache [20] implements a packet processing pipeline for an in-switch read cache based on switch ASICs. DistCache [29] implements distributed in-network caching across multiple racks. The above studies target only read-intensive workloads with write-through caching, which incurs significant overhead under write-intensive workloads (§5). PKache [15] implements in-switch caching with limited associativity and provides a general framework with different cache management policies, yet it does not address write-back caching.

Aside from caching, some studies use programmable switches for efficient storage management. AppSwitch [9] offloads hash-based routing to software switches, and its control plane dynamically updates the routing rules based on server loads for load balancing. NetChain [19] stores records in programmable switches for the coordination of the switch-based chain replication model. TurboKV [13] and Pegasus [26] keep in-switch directory information to speed up the replication protocol of in-memory key-value stores. Concordia [40] tracks the locations of host-side cache copies in programmable switches for efficient cache coherence. Mind [24] maintains in-switch memory management (e.g., address translation and cache coherence) for efficient and transparent rack-scale memory disaggregation. Such systems do not consider in-switch caching for server-side key-value storage.

Write-back caching. Prior studies propose write-back caching policies. DEFER [32] improves the reliability of write-back caching by replication and logging. FlashTier [38] deploys a write-back flash cache and ensures consistency by storing both cached data and mapping details durably in flash. Some studies propose write-back caching policies with different reliability guarantees. Examples include: (i) ordered and journaled policies [23] that provide point-in-time consistency, (ii) write-back flush and persist policies [36] that use write barriers for durable and consistent caching, and (iii) client-side buffered write policies [16] that ensure durability by replication with read-after-write consistency guarantees. However, programmable switches have restricted programming requirements and limited hardware resources for implementing such policies. How to enable new write-back caching policies with stronger reliability guarantees is our future work.

7 Conclusion

FarReach is a fast, available, and reliable in-switch write-back caching framework for load-balanced key-value stores in modern data centers under skewed write-intensive workloads. It incorporates new co-designs of control and data planes for cache admission and eviction under a write-back policy. In particular, FarReach pays special attention to crash-consistent snapshot generation and zero-loss crash recovery, so as to protect against data loss under switch failures. Evaluation under YCSB and synthetic workloads demonstrates the performance benefits of FarReach under skewed write-intensive workloads.

Acknowledgements

We thank our shepherd, Alberto Lerner, and the anonymous reviewers for their comments. This work was supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, Joint Funds of the National Natural Science Foundation of China (U20A20179), National Natural Science Foundation of China (62172007), and Natural Science Foundation of Fujian Province of China (2021J05002). Qun Huang is the corresponding author.

References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, 2012.

[2] 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 *Proc. of USENIX ATC*, 2017.

[3] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proc. of ACM SIGCOMM*, 2013.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In *Proc. of USENIX FAST*, 2020.

[7] 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. *ACM Trans. on Computer Systems*, 26(2):1–26, 2008.

[8] Yue Cheng, Aayush Gupta, and Ali R Butt. An in-memory object caching framework with adaptive load balancing. In *Proc. of ACM EuroSys*, 2015.

[9] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. AppSwitch: Application-layer load balancing within a software switch. In *Proc. of APNet*, 2017.

[10] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

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

[12] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proc. of USENIX NSDI*, pages 79–94, 2019.

[13] Hebatalla Eldakiky, David Hung-Chang Du, and Eman Ramadan. TurboKV: scaling up the performance of distributed key-value stores with in-switch coordination. *CoRR*, abs/2010.14931, 2020.

[14] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proc. of ACM SOCC*, 2011.

[15] Roy Friedman, Or Goaz, and Dor Hovav. Limited associativity caching in the data plane. *CoRR*, abs/2203.04803, 2022.

[16] Shahram Ghandeharizadeh and Hieu Nguyen. Design, implementation, and evaluation of write-back policy with cache augmented data stores. *Proc. of the VLDB Endowment*, 12(8):836–849, 2019.

[17] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proc. of ACM SIGCOMM HotNets Workshop*, 2014.

[18] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of IEEE ICDE*, 2005.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: scale-free sub-RTT coordination. In *Proc. of USENIX NSDI*, 2018.

[20] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP*, 2017.

[21] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *Proc. of WWW*, 2002.

[22] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, pages 654–663, 1997.

[23] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proc. of USENIX FAST*, 2013.

[24] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proc. of ACM SOSP*, 2021.

[25] LevelDB. https://github.com/google/leveldb/.

[26] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proc. of USENIX OSDI*, 2020.

[27] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, cheap and in control with SwitchKV. In *Proc. of USENIX NSDI*, 2016.

[28] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward in-network computation with an in-network cache. In *Proc. of ACM ASPLOS*, 2017.

[29] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: provable load balancing for large-scale storage systems with distributed caching. In *Proc. of USENIX FAST*, 2019.

[30] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: separating keys from values in SSD-conscious storage. *ACM Trans. on Storage*, 13(1):1–28, 2017.

[31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[32] Srivatsan Narasimhan, Sohum Sohoni, and Yiming Hu. A log-based write-back mechanism for cooperative caching. In *Proc. of IEEE IPDPS*, 2003.

[33] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Harry C. Li Herman Lee, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at Facebook. In *Proc. of USENIX NSDI*, 2013.

[34] 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.

[35] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proc. of USENIX ATC*, 2016.

[36] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *Proc. of USENIX ATC*, 2014.

[37] RocksDB. https://github.com/facebook/rocksdb/.

[38] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proc. of ACM EuroSys*, 2012.

[39] Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[40] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: distributed shared memory with in-network cache coherence. In *Proc. of USENIX FAST*, 2021.

[41] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. of USENIX OSDI*, 2020.

[42] YCSB. https://github.com/brianfrankcooper/YCSB/.

CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search

Junhyeok Jang*, Hanjin Choi*†, Hanyeoreum Bae*, Seungjun Lee*, Miryeong Kwon*†, Myoungsoo Jung*†
*Computer Architecture and Memory Systems Laboratory, KAIST
†Panmnesia, Inc.

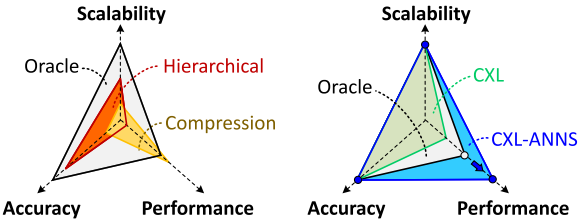
Abstract

We propose *CXL-ANNS*, a software-hardware collaborative approach to enable highly scalable approximate nearest neighbor search (ANNS) services. To this end, we first disaggregate DRAM from the host via compute express link (CXL) and place all essential datasets into its memory pool. While this CXL memory pool can make ANNS feasible to handle billion-point graphs without an accuracy loss, we observe that the search performance significantly degrades because of CXL's far-memory-like characteristics. To address this, CXL-ANNS considers the node-level relationship and caches the neighbors in local memory, which are expected to visit most frequently. For the uncached nodes, CXL-ANNS prefetches a set of nodes most likely to visit soon by understanding the graph traversing behaviors of ANNS. CXL-ANNS is also aware of the architectural structures of the CXL interconnect network and lets different hardware components therein collaboratively search for nearest neighbors in parallel. To improve the performance further, it relaxes the execution dependency of neighbor search tasks and maximizes the degree of search parallelism by fully utilizing all hardware in the CXL network.

Our empirical evaluation results show that CXL-ANNS exhibits 111.1× higher QPS with 93.3% lower query latency than state-of-the-art ANNS platforms that we tested. CXL-ANNS also outperforms an oracle ANNS system that has DRAM-only (with unlimited storage capacity) by 68.0% and 3.8×, in terms of latency and throughput, respectively.

1 Introduction

Dense retrieval (also known as nearest neighbor search) has taken on an important role and provides fundamental support for various search engines, data mining, databases, and machine learning applications such as recommendation systems [1–8]. In contrast to the classic pattern/string-based search, dense retrieval compares the similarity across different objects using their distance and retrieves a given number of objects, similar to the query object, referred to as *k*-nearest neighbor (*kNN*) [9–11]. To this end, dense retrieval embeds input information into a few thousand dimensional spaces of each object, called a feature *vector*. Since these vectors can encode a wide spectrum of data formats (e.g., images, documents, sounds, etc.), dense retrieval understands an input query's semantics, resulting in more context-aware and accurate results than traditional search [6, 12, 13].



(a) Previous studies. (b) CXL-based approaches.

Figure 1: Various billion-scale ANNS characterizations.

Even though kNN is one of the most frequently used search paradigms in various applications, it is a costly operation taking linear time to scan data [14, 15]. This computation complexity unfortunately makes dense retrieval with a billion-point dataset infeasible. To make the kNN search more practical, *approximate nearest neighbor search* (ANNS) restricts a query vector to search only a subset of neighbors with a high chance of being the nearest ones [15–17]. ANNS exhibits good vector searching speed and accuracy, but it significantly increases memory requirement and pressure. For example, many production-level recommendation systems already adopt billion-point datasets, which require tens of TB of working memory space for ANNS; Microsoft search engines (used in Bing/Outlook) require 100B+ vectors, each being explained by 100 dimensions, which consume more than 40TB memory space [18]. Similarly, several of Alibaba's e-commerce platforms need TB-scale memory spaces to accommodate their 2B+ vectors (128 dimensions) [19].

To address these memory pressure issues, modern ANNS techniques leverage lossy compression methods or employ persistent storage, such as solid state disks (SSDs) and persistent memory (PMEM), for their memory expansion. For example, [20–23] split large datasets and group them into multiple clusters in an offline time. This compression approach only has product quantized vectors for each cluster's centroid and searches kNN based on the quantized information, making billion-scale ANNS feasible. On the other hand, the hierarchical approach [24–28] accommodates the datasets to SSD/PMEM, but reduces target search spaces by referring to a summary in its local memory (DRAM). As shown in Figure 1a, these compression and hierarchical approaches can achieve the best kNN search performance and scalabil-

ity similar to or slightly worse than what an *oracle*[1] system offers. However, these approaches suffer from a lack of accuracy and/or performance, which unfortunately hinders their practicality in achieving billion-scale ANNS services.

In this work, we propose *CXL-ANNS*, a software-hardware collaborative approach that enables scalable approximate nearest neighbor search (ANNS). As shown in Figure 1b, the main goal of CXL-ANNS is to offer the latency of billion-point kNN search even shorter than the oracle system mentioned above while achieving high throughput without a loss of accuracy. To this end, we disaggregate DRAM from the host resources via compute express link (CXL) and place all essential datasets into its memory pool; CXL is an open-industry interconnect technology that allows the underlying working memory to be highly scalable and composable with a low cost. Since a CXL network can expand its memory capacity by having more *endpoint devices* (EPs) in a scalable manner, a host's *root-complex* (RC) can map the network's large memory pool (up to 4PB) into its system memory space and use it just like a locally-attached conventional DRAM.

While this CXL memory pool can make ANNS feasible to handle billion-point graphs without a loss of accuracy, we observe that the search performance degrades compared to the oracle by as high as 3.9× (§3.1). This is due to CXL's far-memory-like characteristics; every memory request needs a CXL protocol conversion (from CPU instructions to one or more CXL flits), which takes a time similar to or longer than a DRAM access itself. To address this, we consider the relationship of different nodes in a given graph and cache the neighbors in the local memory, which are expected to visit frequently. For the uncached nodes, CXL-ANNS prefetches a set of nodes most likely to be touched soon by understanding the unique behaviors of the ANNS graph traversing algorithm. CXL-ANNS is also aware of the architectural structures of the CXL interconnect network and allows different hardware components therein to simultaneously search for nearest neighbors in a collaborative manner. To improve the performance further, we relax the execution dependency in the KNN search and maximize the degree of search parallelism by fully utilizing all our hardware in the CXL network.

We summarize the main contribution as follows:

• *Relationship-aware graph caching.* Since ANNS traverses a given graph from its entry-node [10, 19], we observe that the graph data accesses, associated with the innermost edge hops, account for most of the point accesses (§3.2). Inspired by this, we selectively locate the graph and feature vectors in different places of the CXL memory network. Specifically, CXL-ANNS allocates the node information closer to the entry node in the locally-attached DRAMs while placing the other datasets in the CXL memory pool.

• *Hiding the latency of CXL memory pool.* If it needs to traverse (uncached) outer nodes, CXL-ANNS prefetches the datasets of neighbors, most likely to be processed in the next step of kNN queries from the CXL memory pool. However, it is non-trivial to figure out which node will be the next to visit because of ANNS's procedural data processing dependency. We propose a simple foreseeing technique that exploits a unique graph traversing characteristic of ANNS and prefetches the next neighbor's dataset during the current kNN candidate update phase.

• *Collaborative kNN search design in CXL.* CXL-ANNS significantly reduces the time wasted for transferring the feature vectors back and forth by designing EP controllers to calculate distances. On the other hand, it utilizes the computation power of the CXL host for non-beneficial operations in processing data near memory (e.g., graph traverse and candidate update). This collaborative search includes an efficient design of RC-EP interfaces and a sharding method being aware of the hardware configurations of the CXL memory pool.

• *Dependency relaxation and scheduling.* The computation sequences of ANNS are all connected in a serial order, which makes them unfortunately dependent on execution. We examine all the activities of kNN query requests and classify them into urgent/deferrable subtasks. CXL-ANNS then relaxes the dependency of ANN computation sequences and schedules their subtasks in a finer granular manner.

We validate all the functionalities of CXL-ANNS's software and hardware (including the CXL memory pool) by prototyping them using Linux 5.15.36 and 16nm FPGA, respectively. To explore the full design spaces of ANNS, we also implement the hardware-validated CXL-ANNS in gem5 [29] and perform full-system simulations using six billion-point datasets [30]. Our evaluation results show that CXL-ANNS exhibits 111.1× higher bandwidth (QPS) with 93.3% lower query latency, compared to the state-of-the-art billion-scale ANNS methods [20, 24, 25]. The latency and throughput behaviors of CXL-ANNS are even better than those of the oracle system (DRAM-only) by 68.0% and 3.8×, respectively.

2 Background

2.1 Approximate Nearest Neighbor Search

The most accurate method to get k-nearest neighbors (kNN) in a graph is to compare an input query vector with all data vectors in a brute-force manner [9, 31]. Obviously, this simple dense retrieval technique is impractical mainly due to its time complexity [10, 24]. In contrast, approximate nearest neighbor search (ANNS) restricts the query vector to retrieve only a subset of neighbors that can be kNN with a high probability. To meet diverse accuracy and performance requirements, several ANNS algorithms such as tree-structure based [32, 33], hashing based [11, 34, 35] and quantization based approaches [20–23] have been proposed over the past decades. Among the various techniques, ANNS algorithms using graphs [10, 19, 24] are considered as the most promising

[1] In this paper, the term "Oracle" refers to a system that utilizes ample DRAM resources with an unrestricted memory capacity.

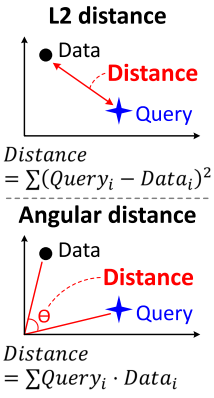


Figure 2: Distance.

Algorithm 1: Best-first Search

Input: *query*,
k := number of nearest neighbor to find
Output: *k* nearest neighbor of *query*

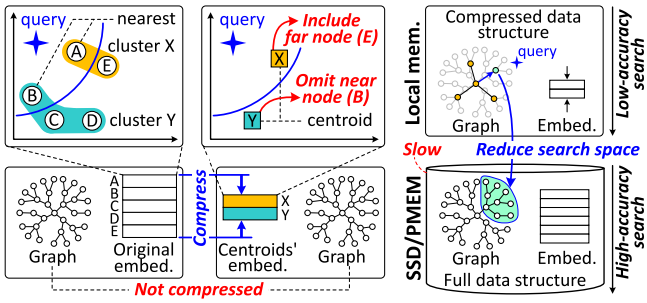
```
1  distance = calcDist(query, startNode)
2  CandidateArr = {startNode, distance}
3  curNode = startNode
4  CandidateArr.markVisited(curNode)
   /* Main iteration (line 5~10) */
5  while !candidateArr.allVisited() do
6      neighbors = Graph.neighbors(curNode)
7      distances = calcDist(query, neighbors)
8      CandidateArr.insert(neighbors, distances)
9      curNode = CandidateArr.getNextNode()
10     CandidateArr.markVisited(curNode)
11 return CandidateArr[: k]
```

solution, with great potential [2]. This is because graph-based approaches can better describe neighbor relationships and traverse fewer points than the other approaches that operate in a Euclidean space [19, 36–39].

Distance calculations. While there are various graph construction algorithms for ANNS [10, 19, 24], the goal of their query search algorithms is all the same or similar to each other; it is simply to find *k* numbers of neighbors in the target graph, which are expected to have the shortest distance from a given feature vector, called *query vector*. There are two most common methods to define such a distance between the query vector and neighbor's feature vector (called *data vector*): i) L2 (Euclidean) distance and ii) angular distance. As shown in Figure 2, these methods map the nodes that we compare into a temporal dimension space using their own vector's feature elements. Let us suppose that there are n numbers of features for each vector. Then, L2 and angular distances are calculated by $\sum_i (Query_i - Data_i)^2$ and $\sum_i (Query_i \cdot Data_i)$, respectively; where $Query_i$ and $Data_i$ are the i^{th} feature of a given query and data vectors, respectively ($i \leq n$). These distance definitions are simplified to reduce their calculation latency, which differs from the actual distances in a multi-dimensional vector space. This simplification works well since ANNS uses the distances only for a relative comparison to search kNN.

Approximate kNN query search. Algorithm 1 explains the graph traversing method that most ANNS employs [10, 19, 24]. The method, best-first search (BFS) [38, 40], traverses from an *entry-node* (line ❸) and moves to neighbors getting closer to the given query vector (lines ❺~❿). While the brute-force search explores a full space of the graph by systematically enumerating all the nodes, ANNS uses a preprocessed graph and visits a limited number of nodes for each hop. The graph is constructed (preprocessed) to have the entry-node that arrives all the nodes of its original graph within the minimum number of average edge hops; this preprocessed graph guarantees that there exists a path between the entry-node and any of the given nodes. To minimize the overhead of graph traverse, BFS employs a *candidate array* that includes the neighbors whose

[2] For the sake of the brevity, we use "graph-based approximate kNN methods" and "ANNS" interchangeably.



(a) Compression-based approach. (b) Hierarchical approach.

Figure 3: Existing billion-scale ANNS methods.

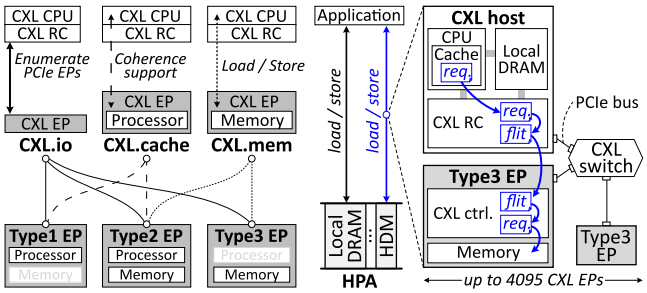
distances (from the query vector) are expected to be shorter than others. For each node visiting, BFS checks this candidate array and retrieves unvisited node from the array (line ❻, ❾). It then calculates the distances of the node's neighbors (line ❼) by retrieving their vectors from the embedding table. After this distance calculation, BFS updates the candidate array with the new information, neighbors and distances (line ❽). All these activities are iterated (line ❺) until there is no unvisited node in the candidate array. BFS finally returns the *k* number of neighbors in the candidate array.

2.2 Towards Billion-scale ANNS

While ANNS can achieve good search speed and reasonable accuracy (as it only visits the nodes in the candidate array), it still requires maintaining all the original graph and vectors in its embedding table. This renders ANNS difficult to have billion-point graphs that exhibit high memory demands in many production-level services [18, 19]. To address this issue, there have been many studies proposed [20–27], but we can classify them into two as shown in Figures 3a and 3b.

Compression approaches. These approaches [20–23] reduce the embedding table by compressing its vectors. As shown in Figure 3a, they logically split the given graph into multiple sub-groups, called *clusters*; the nodes A and E are classified in the cluster X whereas the others are grouped as the cluster Y. For each cluster, these approaches then encode the corresponding vectors into a single, representative vector (called centroid) by averaging all the vectors in the cluster. They then replace all the vectors in the embedding table with their cluster ID. Since the distances are calculated by the compressed centroid vectors (rather than original data vectors), it exhibits a low accuracy for the search. For example, the node E can be selected as one of kNN although the node B sits closer to the query vector. Another issue of these compression approaches is the limited reduction rate in the size of the graph datasets. Since they quantize only the embedding table, their billion-point graph data have no benefit of the compression or even get slightly bigger to add a set of shortcuts into the original graph.

Hierarchical approaches. These approaches [24–27] store all the graph and vectors (embedding table) to the underlying SSD/PMEM (Figure 3b). Since SSD/PMEM are prac-



(a) Types of CXL EP. (b) CXL-based memory pool.

Figure 4: CXL's sub-protocols and endpoint types.

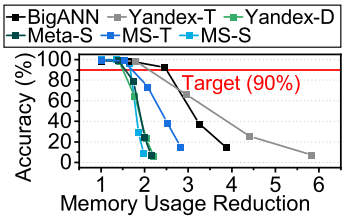


Figure 5: Accuracy.

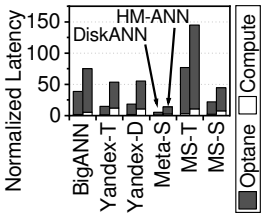


Figure 6: Latency.

tically slower than DRAM by many orders of magnitude, these methods process kNN queries in two separate phases: i) *low-accuracy search* and ii) *high-accuracy search*. The former only refers to compressed or simplified datasets, similar to the datasets that the compression approaches use. The low-accuracy search quickly finds out one or more nearest neighbor candidates (without a storage access) thereby reducing the search space that the latter needs to process. Once it has been completed, the high-accuracy search refers to the original datasets associated with the candidates and processes the actual kNN queries. For example, DiskANN [24] 's low accuracy search finds the kNN candidates using the compressed datasets in DRAM. The high-accuracy search then re-examines and re-ranks the order of kNN candidates by visiting their actual vectors stored in SSD/PMEM. On the other hand, HM-ANN [25] simplifies the target graph by adding several shortcuts (across multiple edge hops) into the graph. HM-ANN's low-accuracy search scans a candidate closer to the given query vector from the simplified graph. Once HM-ANN detects the candidate, the high-accuracy search checks its kNN by referring to all the graph and data vectors recorded in SSD/PMEM.

2.3 Compute Express Link for Memory Pool

CXL is an open standard interconnect which can expand memory over the existing PCIe physical layers in a scalable option [41–43]. As shown in Figure 4a, CXL consists of three sub-protocols: i) *CXL.io*, ii) *CXL.cache*, and iii) *CXL.mem*. Based on which sub-protocols are used for the main communication, CXL EPs can be classified as Types.

Sub-protocols and endpoint types. CXL.io is basically the same as the PCIe standard, which is aimed at enumerating the underlying EPs and performing transaction controls. It is thus used for all the CXL types of EPs to be interconnected to the CXL CPU's root-complex (*RC*) through PCIe. On the other hand, CXL.cache is for an underlying EP to make its states coherent with those of a CXL host CPU, whereas CXL.mem supports simple memory operations (load/store) over PCIe. Type 1 is considered by a co-processor or accelerator that does not have memory exposed to CXL RC while Type 2 employs internal memory, accessible from CXL RC. Thus, Type 1 only uses CXL.cache (in addition to CXL.io), but Type 2 needs to use both CXL.cache and CXL.mem. A potential example of Type 1 and 2 can be FPGAs and GPUs, respectively. On the other hand, Type 3 only uses CXL.mem (read/write), which means that there is no interface for a device-side compute unit to update its calculation results to CXL CPU's RC and/or get a non-memory request from the RC.

CXL endpoint disaggregation. Figure 4b shows how we can disaggregate DRAM from host resources using CXL EPs, in particular, Type 3; we will discuss why Type 3 is the best device type for the design of CXL-ANNS, shortly. Type 3's internal memory is exposed as a *host-managed device memory* (HDM), which can be mapped to the CXL CPU's host physical address (HPA) in the system memory just like DRAM. Therefore, applications running on the CXL CPU can access HDM (EP's internal memory) through conventional memory instructions (loads/stores). Thanks to this characteristic, HDM requests are treated as traditional memory requests in CXL CPU's memory hierarchy; the requests are first cached in CPU cache(s). Once its cache controller evicts a line associated with the address space of HDM, the request goes through to the system's CXL RC. RC then converts one or more memory requests into a CXL packet (called *flit*) that can deal with a request or response of CXL.mem/CXL.cache. RC passes the flit to the target EP using CXL.mem's read or write interfaces. The destination EP's PCIe and CXL controllers take the flit over, convert it to one or more memory requests, and serve the request with the EP's internal memory (HDM).

Type consideration for scaling-out. To expand the memory capacity, the target CXL network can have one or more switches that have multiple ports, each being able to connect a CXL EP. This switch-based network configuration allows an RC to employ many EPs (upto 4K), but only for Type 3. This is because CXL.cache uses virtual addresses for its cache coherence management unlike CXL.mem. As the virtual addresses (brought by CXL flits) are not directly matched with the physical address of each underlying EP's HDM, the CXL switches cannot understand where the exact destination is.

3 A High-level Viewpoint of CXL-ANNS

3.1 Challenge Analysis of Billion-scale ANNS

Memory expansion with compression. While compression methods allow us to have larger datasets, it is not scalable since their quantized data significantly degrades the kNN search accuracy. Figure 5 analyzes the search accuracy of billion-point ANNS that uses the quantization-based com-

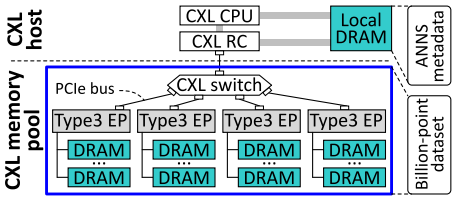


Figure 7: CXL baseline architecture.

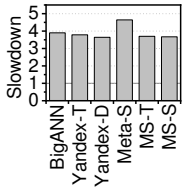


Figure 8: CXL.

pression described in §2.2. In this analysis, it reduces the embedding table by 2×∼16×. We use six billion-point datasets from [30]; the details of these datasets and evaluation environment are the same as what we used in §6.1. As the density of the quantized data vectors varies across different datasets, the compression method exhibits different search accuracies. While the search accuracies are in a reasonable range to service with low compression rates, they significantly drop as the compression rate of the dataset increases. It cannot even reach the threshold accuracy that ANNS needs to support (90%, recommended by [30]) after having 45.8% less data than the original. This unfortunately makes the compression impractical for billion-scale ANNS at high accuracy.

Hierarchical data processing. Hierarchical approaches can overcome this low accuracy issue by adding one more search step to re-rank the results of kNN search. This high-accuracy search however increases the search latency significantly as it eventually requires traversing the storage-side graph and accessing the corresponding data vectors (in storage) entirely. Figure 6 shows the latency behaviors of hierarchical approaches, DiskANN [24] and HM-ANN [25]. In this test, we use 480GB Optane PMEM [44] for DiskANN/HM-ANN and compare their performance with the performance of an oracle ANNS that has DRAM-only (with unlimited storage capacity). One can observe from this figure that the storage accesses of the high-accuracy search account for 87.6% of the total kNN query latency, which makes the search latency of DiskANN and HM-ANN worse than that of the oracle ANNS by 29.4× and 64.6×, respectively, on average.

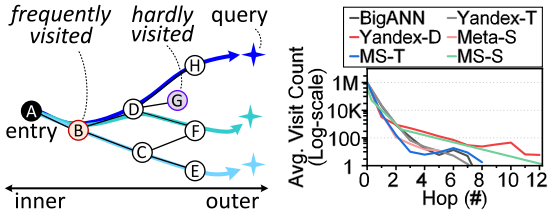
CXL-augmented ANNS. To avoid the accuracy drop and performance depletion, this work advocates to directly have billion-point datasets in a scalable memory pool, disaggregated using CXL. Figure 7 shows our baseline architecture that consists of a CXL CPU, a CXL switch, and four 1TB Type 3 EPs that we prototype (§6.1). We locate all the billion-point graphs and corresponding vectors to the underlying Type 3 EPs (memory pool) while having ANNS metadata (e.g. candidate array) in the local DRAM. This baseline allows ANNS to access the billion-point datasets on the remote-side memory pool just like conventional DRAMs thanks to CXL's instruction-level compatibility. Nevertheless, it is not yet an appropriate option for practical billion-scale ANNS due to CXL's architectural characteristics that exhibit lower performance than the local DRAM.

To be precise, we compare the kNN search latency of the baseline with the oracle ANNS, and the results are shown in Figure 8. In this analysis, we normalize the latency of the baseline to that of the oracle for better understanding. Even though our baseline does not show severe performance depletion like what DiskANN/HM-ANNS suffer from, it exhibits 3.9× slower search latency than the oracle, on average. This is because all the memory accesses associated with HDM(s) insist the host RC convert them to a CXL flit and revert the flit to memory requests at the EP-side. The corresponding responses also requires this *memory-to-flit* conversion in a reverse order thereby exhibiting the long latency for graph/vector accesses. Note that this 3.6∼4.6× performance degradation is not acceptable in many production-level ANNS applications such as recommendation systems [45] or search engines [46].

3.2 Design Consideration and Motivation

The main goal of this work is to make the CXL-augmented kNN search faster than in-memory ANNS services working only with locally-attached DRAMs (cf. CXL-ANNS vs. Oracle as shown in Figure 8). To achieve this goal, we propose CXL-ANNS, a software-hardware collaborative approach, which considers the following three observations: i) node-level relationship, ii) distance calculation, and iii) vector reduction.

Node-level relationship. While there are diverse graph structures [10, 19, 24] for the best-first search traverses (cf. Algorithm 1), all of the graphs starts their traverses from a unique, single entry-node as described in §2.1. This implies that the graph traverse of ANNS visits the nodes closer to the entry-node much more frequently. For example, as shown in Figure 9a, the node B is always accessed to serve a given set of kNN queries targeting other nodes listed in the graph branch while the node G is difficult to visit. To be precise, we examine the average count to visit nodes in all the billion-point graphs that this work evaluate when there are a million kNN query



(a) Traverse. (b) Nodes' visit count.

Figure 9: Graph traverse.

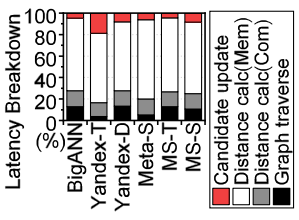
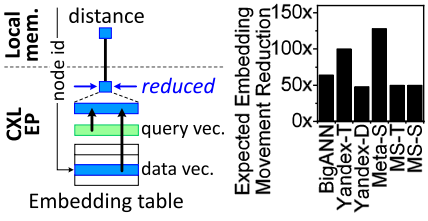


Figure 10: End-to-end breakdown analysis.



(a) Reduction example. (b) Reduction ratio.

Figure 11: Data reduction.

requests. The results are shown in Figure 9b. One can observe from this analysis that the nodes most frequently accessed during the 1M kNN searches reside in the 2∼3 edge hops. By appreciating this node-level relationship, we will locate the graph and vector data regarding inner-most nodes (from the entry-node) to locally-attached DRAMs while allocating all the others to the underlying CXL EPs.

Distance calculation. To analyze the critical path of billion-point ANNS, we decompose the end-to-end kNN search task into four different sub-tasks, i) candidate update, ii) memory access and iii) computing fractions of distance calculation, and iv) graph traverse. We then measure the latency of each sub-tasks on use in-memory, oracle system, which are shown in Figure 10. As can be seen from the figure, ANNS distance calculation significantly contributes to the total execution time, constituting an average of 81.8%. This observation stands in contrast to the widely held belief that graph traversal is among the most resource-intensive operations [47–49]. The underlying reason for this discrepancy is that distance calculation necessitates intensive embedding table lookups to determine the data vectors of all nodes visited by ANNS. Notably, while these lookup operations have the same frequency and pattern as graph traversal, the length of the data vectors employed by ANNS is 2.0× greater than that of the graph data due to their high dimensionality. Importantly, although distance calculation exhibits considerable latency, it does not require substantial computational resources, thus making it a good candidate for acceleration using straightforward hardware solutions.

Reducing data vector transfers. We can take the overhead brought by distance calculations off the critical path in the kNN search by bringing only the distance that ANNS needs to check for each iteration its algorithm visits. As shown in Figure 11a, let's suppose that CXL EPs can compute a distance between a given query vector and data vectors that ANNS is in visit. Since ANNS needs the distance, a simple scalar value, instead of all the full features of each data vector, the amount of data that the underlying EPs transfer can be reduced as many as each vector's dimensional degrees. Figure 11b analyzes how much we can reduce the vector transfers during services of the 1M kNN queries. While the vector dimensions of each dataset varies (96∼256), we can reduce the amount of data to load from the EPs by 73.3×, on average.

3.3 Collaborative Approach Overview

Motivated by the aforementioned observations, CXL-ANNS first caches datasets considering a given graph's inter-node relationship and performs ANNS algorithm-aware CXL prefetches (§4.1). This makes the performance of a naive CXL-augmented kNN search comparable with that of the oracle ANNS. To go beyond, CXL-ANNS reduces the vector transferring latency significantly by letting the underlying EPs to calculate all the ANNS distances near memory (§5.1). As this near-data processing is achieved in a collaborative manner between EP controllers and RC-side ANNS algorithm handler, the performance can be limited by the kNN query service sequences. CXL-ANNS thus schedules kNN search activities in a fine-grained manner by relaxing their execution dependency (§5.3). Putting all together, CXL-ANNS is designed for offering high-performance even better than the oracle ANNS without an accuracy loss.

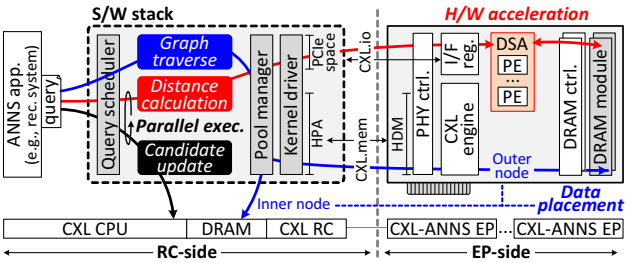


Figure 12: Overview.

Figure 12 shows the high-level viewpoint of our CXL-ANNS architecture, which mainly consists of i) RC-side software stack and ii) EP-side data processing hardware stack.

RC-side software stack. This RC-side software stack is composed of i) query scheduler, ii) pool manager, and iii) kernel driver. At the top of CXL-ANNS, the query scheduler handles all kNN searches requested from its applications such as recommendation systems. It splits each query into three subtasks (graph traverse, distance calculation, and candidate update) and assign them in different places. Specifically, the graph traverse and candidate update subtasks are performed at the CXL CPU side whereas the scheduler allocates the distance calculation to the underlying EP by collaborating with the underlying pool manager. The pool manager handles CXL's HPA for the graph and data vectors by considering edge hop counts, such that it can differentiate graph accesses based on the node-level relationship. Lastly, the kernel driver manages the underlying EPs and their address spaces; it enumerates the EPs and maps their HDMs into the system memory's HPA that the pool manager uses. Since all memory requests for HPA are cached at the CXL CPU, the driver maps EP-side interface registers to RC's PCIe address space using CXL.io instead of CXL.mem. Note that, as the PCIe spaces where the memory-mapped registers exist is in non-cacheable area, the underlying EP can immediately recognize what the host-side application lets the EPs know.

EP-side hardware stack. EP-side hardware stack includes a domain specific accelerator (DSA) for distance calculation in addition to all essential hardware components to build a CXL-based memory expander. At the front of our EPs, a physical layer (PHY) controller and CXL engine are implemented, which are responsible for the PCIe/CXL communication control and flit-to-memory request conversion, respectively. The converted memory request is forwarded to the underlying memory controller that connects multiple DRAM modules at its backend; in our prototype, an EP has four memory controllers, each having a DIMM channel that has 256GB DRAM

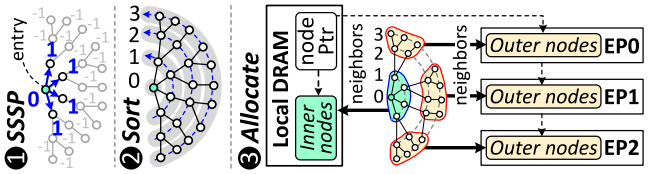


Figure 13: Data placement.

modules. On the other hand, the DSA is located between the CXL engine and memory controllers. It can read data vectors using the memory controllers while checking up the operation commands through CXL engine's interface registers. These interface registers are mapped to the host non-cacheable PCIe space such that all the commands that the host writes can be immediately visible to DSA. DSA calculates the approximate distance for multiple data vectors using multiple processing elements (PEs), each having simple arithmetic units such as adder/subtractor and multiplier.

4 Software Stack Design and Implementation

From the memory pool management viewpoint, we have to consider two different system aspects: i) graph structuring technique for the local memory and ii) efficient space mapping method between HDM and graph. We will explain the design and implementation details of each method in this section.

4.1 Local Caching for Graph

Graph construction for local caching. While the pool manager allocates most graph data and all data vectors to the underlying CXL memory pool, it caches the nodes, expected to be most frequently accessed, in local DRAMs as much as the system memory capacity can accommodate. To this end, the pool manager considers how many edge hops (i.e., calculating the number of edge hops) exist from the fixed entry-node to each node for its relationship-aware graph cache. Figure 13 explains how the pool manager allocates the nodes in a given graph to different places (local memory vs. CXL memory pool). When constructing the graph, the pool manager calculates per-node hop counts by leveraging a single source shortest path (SSSP) algorithm [50, 51]; it first lets all the nodes in the graph have a negative hop count (e.g., -1). Starting from the entry-node, the pool manager checks all the nodes in one edge hop and increases its hop count. It visits each of the nodes and iterates this process for them until there is no node to visit in a breadth-first search manner. Once each node has its own hop count, the pool manager sorts them based on the hop count in an ascending order and allocates the nodes from the top (having the smallest hop count) to local DRAMs as many as it can. The available size of the local DRAMs can be simply estimated by referring to system configuration variables (`sysconf()`) of the total number of pages (`_SC_AVPHYS_PAGES`) and the size of each page (`_SC_PAGESIZE`). It's important to mention that in this study, the pool manager uses several threads within the user space to execute SSSP, aiming to reduce the construction time

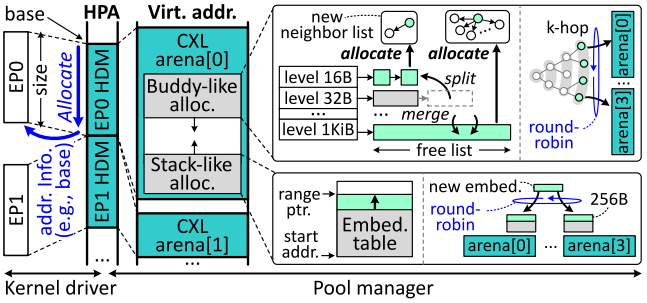


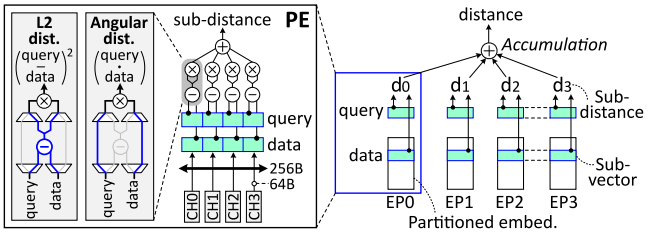
Figure 14: Memory management.

to a minimum. Once the construction is done, the threads are terminated to make sure they do not consume CPU resources when a query is given.

4.2 Data Placement on the CXL Memory Pool

Preparing CXL for user-level memory. When mapping HDM to the system memory's HPA, CXL CPU should be capable of recognizing different HDMs and their size whereas each EP needs to know where its HDM is assigned in HPA. As shown in Figure 14, our kernel driver checks PCIe configuration space and figures out CXL devices at the PCIe enumeration time. The driver then checks RC information from the system's data structure describing the hardware components that show where the CXL HPA begins (base), such as device tree [52] or ACPI [53]. From the base, our kernel driver allocates each HDM as much as it defines in a contiguous space. It lets the underlying EPs know where each of corresponding HDM is mapped in HPA, such that they can convert the address of memory requests (HPA) to its original HDM address. Once all the HDMs are successfully mapped to HPA, the pool manager allocates each HDM to different places of user-level virtual address space that the query scheduler operates on. This memory-mapped HDM, called *CXL arena*, guarantees per-arena continuous memory space and allows the pool manager to distinguish different EPs at the user-level.

Pool management for vectors/graph. While CXL arenas directly expose the underlying HDMs of CXL EPs to user-level space, it should be well managed to accommodate all the billion-point datasets appreciating their memory usage behaviors. The pool manager considers two aspects of the datasets; the data vectors (i.e., embedding table) should be located in a substantially large and consecutive memory space while the graph structure requires taking many neighbor lists with variable length (16B∼1KB). The pool manager employs stack-like and buddy-like memory allocators, which grow upward and downward in each CXL arena, respectively. The former allocator has a range pointer and manages memory for the embedding table, similar to stack. The pool manager allocates the data vectors across multiple CXL arenas in a round-robin manner by considering the underlying EP architecture. This vector sharding method will be explained in §5.1. In contrast, the buddy-like allocator employs a level pointer,



(a) PE architecture. (b) Sharding.

Figure 15: Distance calculation.

each level consisting of a linked list, which connects data chunks with different size (from 16B to 1KB). Like Linux buddy memory manager [54], it allocates the CXL memory spaces as much as each neighbor list exactly requires and merge/split the chunk(s) based on the workload behaviors. To make each EP balanced, the pool manager allocates the neighbor lists for each hop in round-robin manner across different CXL arenas.

5 Collaborative Query Service Acceleration

5.1 Accelerating Distance Calculation

Distance computing in EP. As shown in Figure 15a, a processing element (PE) of DSA has arithmetic logic tree connecting a multiplier and subtractor at each terminal for element-wise operations. Depending on how the dataset's features are encoded, the query and data vectors are routed differently to the two units as input. If the features are encoded for the Euclidean space, the vectors are supplied to the subtractor for L2 distance calculation. Otherwise, the multiplexing logics directly deliver the input vectors to the multiplier by bypassing the subtractor such that it can calculate the angular distance. Each terminal simultaneously calculates individual elements of the approximate distance, and the results are accumulated by going through the arithmetic logic tree network from the terminal to its root. In addition, each PE's terminal reads data from all four different DIMM channels in parallel, thus maximizing the EP's backend DRAM bandwidth.

Vector sharding. Even though each EP has many PEs (10 in our prototype), if we locate the embedding table from the start address of an EP in a consecutive order, EP's backend DRAM bandwidth can be bottleneck in our design. This is because each feature vector in the embedding table is encoded by high dimensional information (∼256 dimensions, taking around 1KB). To address this, our pool manager shards the embedding table in a column wise and stores different parts of the table across the different EPs. As shown in Figure 15b, this *vector sharding* splits each vector into multiple sub-vectors based on each EP's I/O granularity (256B). Each EP simultaneously computes its sub-distance from the split data vector that the EP accommodates. Later, the CXL CPU accumulates the sub-distances to get the final distance value. Note that, since the L2 and angular distances are calculated by accumulating the output of element-wise operations, the final distance is the same as the results of the sub-distance accumulation using vector sharding.

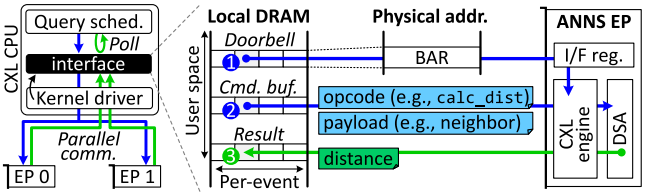


Figure 16: Interface.

Interfacing with EP-level acceleration. Figure 16 shows how the interface registers are managed to let the underlying EPs compute a distance where the data vectors exist. There are two considerations for the interface design and implementation. First, multiple EPs perform the distance calculation for the same neighbors in parallel thanks to vector sharding. While the neighbor list contains many node ids (≤200), it is thus shared by the underlying EPs. Second, handling interface registers using CXL.io is an expensive operation as CPU should be involved in all the data copies. Considering these two, the interface registers handle only the event of command arrivals, called *doorbell* whereas each EP's CXL engine pulls the corresponding operation type and neighbor list from the CPU-side local DRAM (called a command buffer) in an active manner. This method can save the time for CPU to move the neighbor list to each EP's interface registers one by one as the CXL engine brings all the information if there is any doorbell update. The CXL engine also pushes results of distance calculation to the local DRAM such that the RC-side software directly accesses the results without an access of the underlying CXL memory pool. Note that all these communication buffers and registers are directly mapped to the user-level virtual addresses in our design such that we can minimize the number of context switches between user and kernel mode.

5.2 Prefetching for CXL Memory Pool

Figure 17a shows our baseline of collaborative query service acceleration, which lets EPs compute sub-distances while ANNS's graph traverse and candidate update (including sub-distance accumulation) are handled at the CPU-side. This scheduling pattern is iterated until there is no kNN candidates to visit futher (Algorithm 1). A challenge of this baseline approach is traversing graph can be started once the all node information is ready at the CPU-side. While local caching

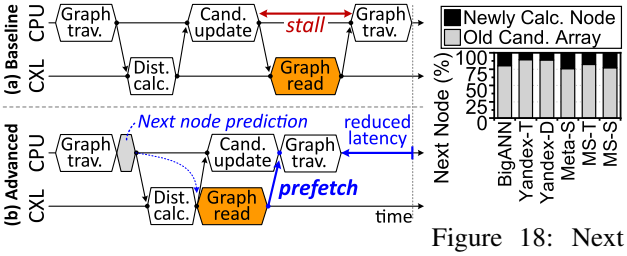


Figure 17: Prefetching.

Figure 18: Next node's source.

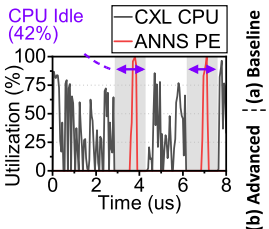


Figure 18: Resource utilization.

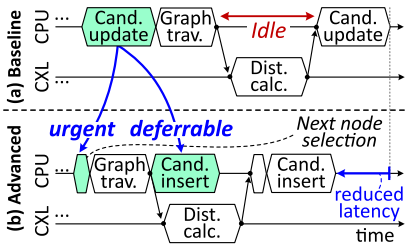


Figure 19: Query scheduling.

of our pool manager addreses this, it yet shows a limited performance. It is required to go through the CXL memory pool to get nodes, which does not sit in the innermost edge hops. As its latency to access the underlying memory pool is long, the graph traverse can be postponed comparably. To this end, our query scheduler prefetches the graph information earlier than the actual traverse subtask needs, as shown in Figure 17b.

While this prefetch can hide the long latency imposed by the CXL memory pool accesses, it is non-trivial as the prefetch requires knowing the nodes that the next (future) iteration of the ANNS algorithm will visit. Our query scheduler speculates the nodes to visit and brings their neighbor information by referring to the candidate array, which is inspired by an observation that we have. Figure 18 shows which nodes are accessed in the graph traverse of the next iteration across all the datasets that we tested. We can see that 82.3% of the total visiting nodes are coming from the candidate array (even though its information is not yet updated for the next step).

5.3 Fine-Granular Query Scheduling

Our collaborative search query acceleration can reduce the amount of data to transfer significantly and successfully hide the long latency imposed by the CXL memory pool. However, computing kNN search in different places makes the RC-side ANNS subtasks pending until EPs complete their distance calculation. Figure 18 shows how much the RC-side subtasks (CXL CPU) stay idle, waiting for the distance results. In this evaluation, we use Yandex-D as a representative of the datasets, and its time series are analyzed for the time visiting only first two nodes for their neighbor search. The CXL CPU performs nothing while EPs calculate the distances, which take 42% of the total execution time for processing those two nodes. This idle time cannot be easily removed as candidate update cannot be processed without having their distance.

To address this, our query scheduler relaxes the execution dependency on the candidate update and separates such an update into urgent and deferrable procedures. Specifically, the candidate update consists of i) inserting (updating) the array with the candidates, ii) sorting kNN candidates based on their distance, and iii) node selection to visit. The node selection is an important process because the following graph traverse requires knowing the nodes to visit (urgent). However, sorting/inserting kNN candidates maintain the k numbers of neighbors in the candidate array, which are not to be done immediately. Thus, as shown in Figure 19, the query scheduler performs the node selection before the graph traverse, but it executes the deferrable operations during the distance calculation time by delaying them in a fine-granular manner.

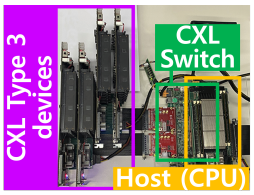


Figure 21: Prototype.

CPU	40 O3 cores, ARM v8, 3.6GHz L1/L2 $: 64KiB/2MiB per core
Local memory	128GiB, DDR4-3200
CXL memory pool	1 CXL switch 256GiB/device, DDR4-3200
Storage	4× Intel Optane 900P 480 GB
CXL-ANNS	1 GHz, 10 ANNS PE/device, 2 distance calc. unit/PE

Table 1: Simulation setup.

6 Evaluation

6.1 Evaluation Setup

Prototype and Methodology. Given the lack of a publicly available, fully functional CXL system, we constructed and validated the CXL-ANNS software and hardware in an operational real system (Figure 21). This hardware prototype is based on a 16nm FPGA. To develop our CXL CPU prototype, we adapted the RISC-V CPU [55]. The prototype integrates 4 ANNS EPs, each equipped with four memory controllers, linked to the CXL CPU via a CXL switch. The system's software for the prototype, including the kernel driver, is compatible with Linux 5.15.36. For the ANNS execution, we adjusted Meta's open ANNS library, FAISS v1.7.2 [56].

Unfortunately, the prototype system does not offer the flexibility needed to explore various ANNS design spaces. As a remedy, we also established a hardware-validated full-system simulator [29] that represents CXL-ANNS, which was utilized for evaluation. This model replicates all operational cycles extracted from the hardware prototype and is cross-validated with our real system at the cycle level. We conducted simulation-based studies in this evaluation, the system details of which are outlined in Table 1. Notably, the system emulates the server utilized in Meta's production environment [57]. Although our system by default uses 4 EPs, our system increases their count for specific workloads (e.g., Meta-S) that necessitate larger memory spaces (more than 2TB) compared to others.

Workloads. We use billion-scale ANNS datasets from BigANN benchmark [30], a public ANNS benchmark that multiple companies (e.g., Microsoft, Meta) participate in. Their important characteristics are summarized in Table 2. In addition, since ANNS-based services often need different number

Dataset	Dist.	Num. vecs.	Emb. dim.	Avg. num. neighbors	Candidate arr. size k=1	k=5	k=10	Num. devices.
BigANN	L2	1B	128	31.6	30	75	150	4
Yandex-T	Ang.	1B	200	29.0	440	900	2500	4
Yandex-D	L2	1B	96	66.9	300	700	1700	4
Meta-S	L2	1B	256	190	1200	2800	5600	8
MS-T	L2	1B	100	43.1	60	130	250	4
MS-S	L2	1B	100	87.4	580	100	200	4

Table 2: Workloads.

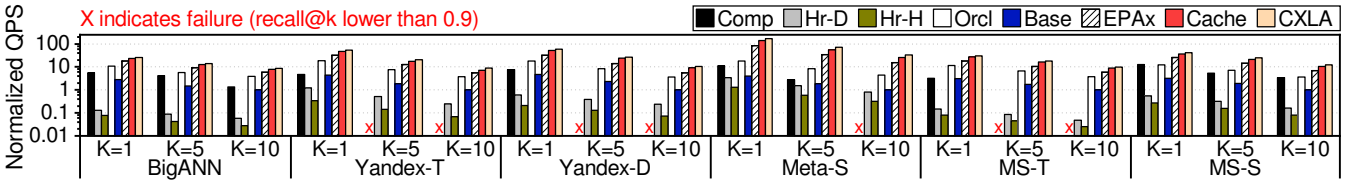
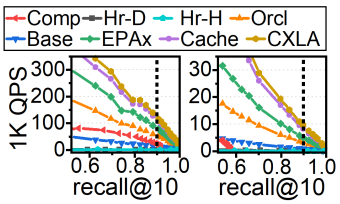


Figure 22: Throughput (queries per second).



(a) BigANN. (b) Yandex-D.

Figure 23: Recall-QPS curve.

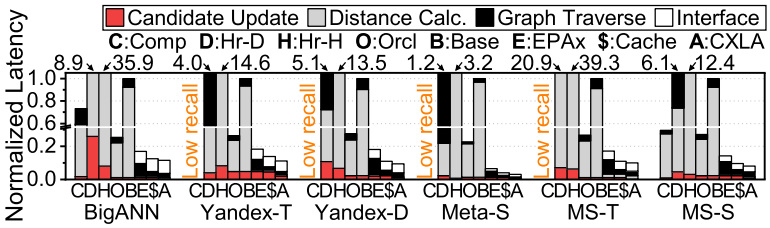


Figure 24: Single query latency (k = 10).

Dataset	Base	CXL-ANNS
BigANN	3.0	0.3
Yandex-T	66.0	7.4
Yandex-D	55.7	5.3
Meta-S	1121.2	34.2
MS-T	6.0	0.6
MS-S	107.2	8.6

* unit: ms

Table 3: Latency.

of nearest neighbors [3, 19], we evaluated our system on the various k (e.g., 1, 5, 10). We generated the graph for ANNS by using state-of-the-art algorithm, NSG, employed by production search services in Alibaba [19]. Since the accuracy and performance of BFS can vary on the size of the candidate array, we only show the performance behavior of our system when its accuracy is 90%, as recommended by the BigANN benchmark. The accuracy is defined as *recall@k*; the ratio of the exact k number of neighbors that are included in the k number of output nearest neighbors of ANNS.

Configurations. We compare CXL-ANNS with 3 state-of-the-art large-scale ANNS systems. For the compression approach, we use a representative algorithm, product quantization [20] (***Comp***). It compresses the data vector by replacing the vector with the centroid of its closest cluster (see §2.2). For the hierarchical approach, we use DiskANN [24] (***Hr-D***) and HM-ANN [25] (***Hr-H***) for the evaluation. The two methods employ compressed embedding table and simplified graphs to reduce the number of SSD/PMEM accesses, respectively. For fair comparison, we use the same storage device, Intel Optane [44], for both Hr-D/H. For CXL-ANNS, we evaluated its multiple variants to distinguish the effect of each method we propose. Specifically, ***Base*** places the graph and embedding table in CXL memory pool and lets CXL CPU execute the subtasks of ANNS. Compared to Base, ***EPAx*** performs distance calculation by using DSA inside the ANNS EP. Compared to EPAx, ***Cache*** employs relationship-aware graph caching and prefetching. Lastly, ***CXLA*** employs all the methods we propose, including fine-granular query scheduling. In addition, we compare oracle system (***Orcl***) that uses unlimited local DRAM. We will show that CXL-ANNS makes the CXL-augmented kNN search faster than Orcl.

6.2 Overall Performance

We first compare the throughput and latency of various systems we evaluated. We measured the systems' throughput by counting the number of processed queries per second (QPS, in short). Figure 22 shows the QPS of all ks, while Figure 24 digs the performance behavior deeper for k=10 by breaking down the latency. We chose k=10 following the guide from BigANN benchmark. The performance behavior for k=1,5 are largely same with when k=10. For both figures, we normalized the values by that of *Base* when k=10. The original latencies are summarized in Table 3.

As shown in Figure 22, the QPS gets lower when the k increases for all the systems we tested. This is because, the BFS visits more nodes to find more nearest neighbors. On the other hand, while Comp exhibits comparable QPS to Orcl, it fails to reach the target recall@k (0.9) for 7 workloads. This is because Comp cannot calculate the exact distance since it replaces the original vector with the centroid of a cluster nearby. This can also be observed in Figure 23. The figure shows the accuracy and QPS when we vary the size of candidate array for two representative workloads. BigANN represents the workloads that Comp does reach the target recall, while Yandex-D represents the opposite. We can see that Comp converges at low recall@10 of 0.92 and 0.58, respectively, while other systems reach the maximum recall@10.

In contrast, hierarchical approaches (Hr-D/H) reaches the target recall@k for all the workloads we tested, by re-ranking the search result. However, they suffer from the long latency of underlying SSD/PMEM while accessing their uncompressed graph and embedding table. Such long latency significantly depletes the QPS of Hr-D and Hr-H by 35.9× and 77.6× compared to Orcl, respectively. Consider Figure 24 to better understand; Since Hr-D only calculates the distance for the limited number of nodes in the candidate array, it exhibits 20.1× shorter distance calculation time compared to Hr-H, which starts a new BFS on original graph stored in SSD/PMEM. However, Hr-D's graph traverse takes longer time than that of Hr-H by 16.6×. This is because, Hr-D accesses the original graph in SSD/PMEM for both low/high-accuracy search while Hr-H accesses the original graph only for their high-accuracy search.

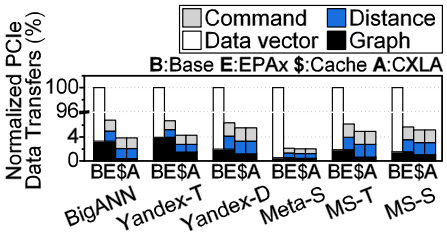


Figure 25: Data transfer.

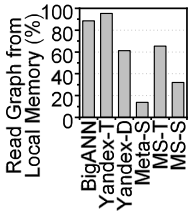


Figure 26: Local caching.

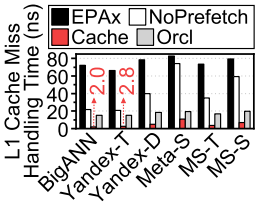


Figure 27: Cache miss handling time.

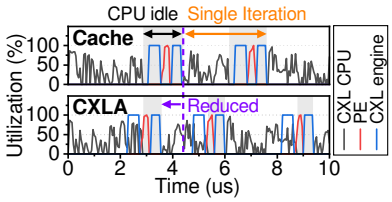


Figure 28: CPU/PE utilization (Yandex-D).

As shown in Figure 22, Base does not suffer from accuracy drop or the performance depletion of Hr-D/H since it employs a scalable memory pool that CXL offers. Therefore, it significantly improves the QPS by 9.4× and 20.3×, compared to Hr-D/H, respectively. However, Base still exhibits 3.9× lower throughput than Orcl. This is because Base experiences the long latency of memory-to-flit conversion while accessing the graph/embedding in CXL memory pool. Such conversion makes Base's graph traverse and distance calculation longer by 2.6× and 4.3×, respectively, compared to Orcl.

Compared to Base, EPAx significantly diminishes the distance calculation time by a factor of 119.4×, achieved by reducing data vector transfer through the acceleration of distance calculation within the EPs. While this EP-level acceleration introduces an interface overhead, this overhead only represents 5.4% of the Base's distance calculation latency. Hence, EPAx reduces the query latency by 7.5× on average, relative to Base. It's important to highlight that EPAx's latency is 1.9× lower than Orcl's, which has unlimited DRAM. This discrepancy stems from Orcl's insufficient grasp of the ANNS algorithm and its behaviour, which results in considerable data movement overhead during data transfer between local memory and the processor complex. Additional details can be found in Figure 25, depicting the volume of data transfer via PCIe for the CXL-based systems. The figure shows that EPAx eliminates data vector transfer, thereby cutting down data transfer by 21.1×.

Further, Cache improves EPAx's graph traversal time by 3.3×, thereby enhancing the query latency by an average of 32.7%. This improvement arises because Cache retains information about nodes anticipated to be accessed frequently in the local DRAM, thereby handling 59.4% of graph traversal within the local DRAM (Figure 26). The figure reveals a particularly high ratio for BigANN and Yandex-T, at 92.0%. As indicated in Table 2, their graphs have a relatively small number of neighbors (31.6 and 29.0, respectively), resulting in their graphs being compact at an average of 129.3GB. In contrast, merely 13.8% of Meta-S's graph accesses are serviced from local memory, attributable to its extensive graph. Nevertheless, even for Meta-S, Cache enhances graph traversal performance by prefetching graph information before actual visitation. As depicted in Figure 24, this prefetching can conceal CXL's prolonged latency, reducing Meta-S's graph traversal latency by 72.8%. While prefetching would introduce overhead in speculating the next node visit, it is insignificant, accounting for only 1.3% of the query latency. These caching and prefetching techniques yield graph processing performance similar to that of Orcl. We will explain the details of prefetching shortly.

Lastly, as depicted in Figure 22, CXLA boosts the QPS by 15.5% in comparison to Cache. This is due to CXLA's enhancement of hardware resource utilization by executing deferrable subtasks and distance calculations concurrently in the CXL CPU and PE, respectively. As illustrated in Figure 24, such scheduling benefits Yandex-T, Yandex-D, and Meta-S more so than others. This is attributable to their use of a candidate array that is, on average, 16.3× larger than others, which allows for the overlap of updates with distance calculation time. Overall, CXLA attains a significantly higher QPS than Orcl, surpassing it by an average factor of 3.8×.

6.3 Collaborative Query Service Analysis

Prefetching. Figure 27 compares the L1 cache miss handling latency while accessing the graph for the CXL-based systems we tested. We measured the latency by dividing the total L1 cache miss handling time of CXL CPU by the number of L1 cache access. The new system, NoPrefetch, disables the prefetching from Cache. As shown in Figure 27, EPAx's latency is as long as 75.4ns since it accesses slow CXL memory whenever there is a cache miss. NoPrefetch alleviates such problem thanks to local caching, shortening the latency by 45.8%. However, when the dataset uses a large graph (e.g. Meta-S, MS-S), only 24.5% of the graph can be cached in local memory. This makes NoPrefetch's latency 2.3× higher than that of Orcl. In contrast, Cache significantly shortens the latency by 8.5× which is even shorter than that of Orcl. This is because Cache can foresee the next visiting nodes and loads the graph information in the cache in advance. Note that, Orcl accesses local DRAM on demand on cache miss.

Utilization. Figure 28 shows the utilization of CXL CPU, PE, CXL engine on a representative dataset (Yandex-D). To clearly provide the behavior of our fine-granule scheduling, we composed a CXL-ANNS with single-core CXL CPU and single PE per device and show their behavior in a timeline. The upper part of the figure shows the behavior of Cache that does not employ the proposed scheduling. We plot CXL CPU's utilization as 0 when it polls the distance calculation results of PE, since it does not perform any useful job during that time. As shown in the figure, CXL CPU idles for 42.0% of the total time waiting for the distance calculation result. In

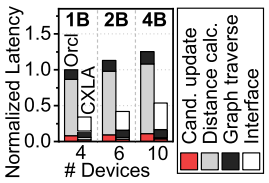


Figure 29: Device scaling (Yandex-D).

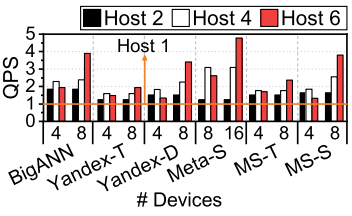


Figure 30: Host Sensitivity.

contrast, CXLA reduces the idle time by 1.3×, relaxing the dependency between ANNS subtasks. In the figure, we can see that the CXL CPU's candidate update time overlaps with the time CXL engine and PE handling the command. As a result, CXLA improves the utilization of hardware resources in CXL network by 20.9%, compared to Cache.

6.4 Scalability Test

Bigger Dataset. To evaluate the scalability of CXL-ANNS, we increase the number of data vectors in Yandex-D by 4B, and connect more EP to CXL CPU to accommodate their data. Since there is no publicly available dataset that is as large as 4B, we synthetically generated additional 3B vectors by adding noise to original 1B vectors. As shown in Figure 29, we can see that the latency of Orcl increases as we increase the scale of dataset. This is because larger dataset makes BFS visit more nodes to maintain the same level of recall. On the other hand, we can see the interface overhead of CXLA increases as we employ more devices to accommodate bigger dataset. This is because the CXL CPU should notify more devices for the command arrival by ringing the doorbell. Despite such overhead, CXLA exhibits 2.7× lower latency than Orcl thanks to its efficient collaborative approach.

Multi-host. In a disaggregated system, a natural way to increase the system's performance is to employ more host CPUs. Thus, we evaluate the CXL-ANNS that supports multiple hosts in the CXL network. Specifically, we split EP's resources such as HDM and PEs and then allocate each of them to one the the CXL hosts in the network. For ANNS, we partition the embedding table and make each host responsible for finding kNN from different partitions. Once all the CXL hosts find the kNN, the system gathers them all and rerank the neighbors to finally select kNN among them.

Figure 30 shows the QPS of multi-host ANNS. The QPS is normalized to that when we use single CXL host with the same number of EPs that we used before. Note that we also show the QPS when we employ more number of EPs than we used before. When the number of EPs stays the same, the QPS increases until we connect 4 CXL hosts in the system. However the QPS drops when the number of CXL hosts is 6. This is because the distance calculation by limited number of PEs became the bottleneck; the commands from the host pends since there is no availabe PE. Such problem can be addressed by having more EPs in the system, thereby distributing the computation load. As we can see in the figure, when we double the number of EPs in the network, we can improve the QPS when we have 6 CXL hosts in the system.

7 Discussion and Acknowledgments

GPU-based distance calculation. Recent research has begun to leverage the massive parallel processing capabilities of GPUs to enhance the efficiency of graph-based ANNS services [58, 59]. While GPUs generally exhibit high performance, our argument is that it's not feasible for CPU+GPU memory to handle the entirety of ANNS data and tasks, as detailed in Section 1. Even under the assumption that ANNS is functioning within an optimal in-memory computing environment, there are two elements to consider when delegating distance computation to GPUs. The first point is that GPUs require interaction with the host's software and/or hardware layers, which incurs a data transfer overhead for computation. Secondly, ANNS distance computations can be carried out using a few uncomplicated, lightweight vector processing units, making GPUs a less cost-efficient choice for these distance calculation tasks.

In contrast, CXL-ANNS avoids the burden of data movement overhead, as it processes data in close proximity to its actual location and returns only a compact result set. This approach to data processing is well established and has been validated through numerous application studies [48, 60–66]. Moreover, CXL-ANNS effectively utilizes the cache hierarchy and can even decrease the frequency of accesses to the underlying CXL memory pool. It accomplishes this through its CXL-aware and ANNS-aware prefetching scheme, which notably enhances performance.

Acknowledgments. The authors thank anonymous reviewers for their constructive feedback as well as Panmnesia for their technical support. The authors also thank Sudarsun Kannan for shepherding this paper. This work is supported by Panmnesia and protected by one or more patents. Myoungsoo Jung is the corresponding author (mj@camelab.org)

8 Conclusion

We propose CXL-ANNS, a software-hardware collaborative approach for scalable ANNS. CXL-ANNS places all the dataset into its CXL memory pool to handle billion-point graphs while making the performance of the kNN search comparable with that of the (local-DRAM only) oracle system. To this end, CXL-ANNS considers inter-node relationship and performs ANNS-aware prefetches. It also calcualate distances in its EP while scheduling the ANNS subtasks to utilize all the resources in the CXL network. Our empirical results show that CXL-ANNS exhibits 111.1× better performance compared to the state-of-the-art billion-scale ANNS methods and 3.8× better performance than oracle system, respectively.

References

[1] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jun Jiang, Yunlong Xu, Hongbo Deng, et al. Approximate nearest neighbor search under neural similarity metric for large-scale recommendation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM)*, 2022.

[2] Yanhao Zhang, Pan Pan, Yun Zheng, Kang Zhao, Yingya Zhang, Xiaofeng Ren, and Rong Jin. Visual search at alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

[3] Jianjin Zhang, Zheng Liu, Weihao Han, Shitao Xiao, Ruicheng Zheng, Yingxia Shao, Hao Sun, Hanqing Zhu, Premkumar Srinivasan, Weiwei Deng, et al. Uni-retriever: Towards learning the unified embedding based retriever in bing sponsored search. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2022.

[4] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021.

[5] Minjia Zhang and Yuxiong He. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*, 2019.

[6] 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 (KDD)*, 2020.

[7] Jiawen Liu, Zhen Xie, Dimitrios Nikolopoulos, and Dong Li. RIANN: Real-time incremental learning with approximate nearest neighbor on mobile devices. In *2020 USENIX Conference on Operational Machine Learning (OpML 20)*, 2020.

[8] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, et al. Autosys: The design and operation of learning-augmented systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 323–336, 2020.

[9] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, 2008.

[10] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[11] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[12] Pandu Nayak. Understanding searches better than ever before. https://blog.google/products/search/search-language-understanding-bert/, 2019.

[13] Charlie Waldburger. As search needs evolve, microsoft makes ai tools for better search available to researchers and developers. https://news.microsoft.com/source/features/ai/bing-vector-search/, 2019.

[14] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[15] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998.

[16] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6), 1998.

[17] Ting Liu, Andrew Moore, Ke Yang, and Alexander Gray. An investigation of practical approximate nearest neighbor algorithms. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems (NIPS)*, 2004.

[18] Harsha Simhadri. Research talk: Approximate nearest neighbor search systems at scale. https://www.youtube.com/watch?v=BnYNdSIKibQ&list=PLD7HFcN7LXReJTWFKYqwMcCc1nZKIXBo9&index=9, 2021.

[19] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 2019.

[20] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE*

Transactions on Pattern Analysis and Machine Intelligence, 2010.

[21] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning (ICML)*, 2020.

[22] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2013.

[23] Artem Babenko and Victor Lempitsky. The inverted multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[24] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[25] Jie Ren, Minjia Zhang, and Dong Li. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[26] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. Freshdiskann: A fast and accurate graph-based ANN index for streaming similarity search. *arXiv preprint arXiv:2105.09613*, 2021.

[27] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, et al. Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In *Proceedings of the ACM Web Conference 2023 (WWW 23)*, 2023.

[28] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighbor search. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[29] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[30] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 2022.

[31] Sunil Arya and David M Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, 1993.

[32] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[33] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 36(2):388–403, 2013.

[34] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 2015.

[35] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment*, 2014.

[36] 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*. Springer, 2017.

[37] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. Return of the lernaean hydra: experimental evaluation of data series approximate similarity search. *Proceedings of the VLDB Endowment*, 2019.

[38] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631*, 2021.

[39] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[40] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[41] CXL Consortium. Compute express link 3.0 white paper. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf, 2022.

[42] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.

[43] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

[44] Intel. Optane ssd 9 series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html, 2021.

[45] Michael Anderson, Benny Chen, Stephen Chen, Summer Deng, Jordan Fix, Michael Gschwind, Aravind Kalaiah, Changkyu Kim, Jaewon Lee, Jason Liang, et al. First-generation inference accelerator deployment at facebook. *arXiv preprint arXiv:2107.04140*, 2021.

[46] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013.

[47] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[48] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[49] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020.

[50] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[51] Andy Diwen Zhu, Xiaokui Xiao, Sibo Wang, and Wenqing Lin. Efficient single-source shortest path and distance queries on large graphs. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013*. ACM, 2013.

[52] Linaro. The devicetree specification. .https://www.devicetree.org/.

[53] Inc UEFI Forum. Advanced configuration and power interface (acpi) specification version 6.4. https://uefi.org/specs/ACPI/6.4/, 2021.

[54] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 1965.

[55] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson, and Krste Asanovic. Boomv2: an open-source out-of-order risc-v core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[56] Herve Jegou, Matthijs Douze, Jeff Johnson, Lucas Hosseini, Chengqi Deng, and Alexandr Guzhva. Faiss. https://github.com/facebookresearch/faiss, 2018.

[57] 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)*, pages 488–501. IEEE, 2020.

[58] Weijie Zhao, Shulong Tan, and Ping Li. Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020.

[59] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Gpu-accelerated proximity graph approximate nearest neighbor search and construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022.

[60] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/software co-programmable

framework for computational SSDs to accelerate deep learning service on large-scale graphs. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

[61] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[62] Miryeong Kwon, Junhyeok Jang, Hanjin Choi, Sangwon Lee, and Myoungsoo Jung. Failure tolerant training with persistent memory disaggregation over cxl. *IEEE Micro*, 2023.

[63] Jun Heo, Seung Yul Lee, Sunhong Min, Yeonhong Park, Sung Jun Jung, Tae Jun Ham, and Jae W Lee. Boss: Bandwidth-optimized search accelerator for storage-class memory. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021.

[64] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

[65] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, et al. Genstore: a high-performance in-storage processing system for genome sequence analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[66] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, et al. A case study of processing-in-memory in off-the-shelf systems. In *USENIX Annual Technical Conference (ATC)*, 2021.

Overcoming the Memory Wall with CXL-Enabled SSDs

Shao-Peng Yang
Syracuse University

Minjae Kim
DGIST

Sanghyun Nam
Soongsil University

Juhyung Park
DGIST

Jin-yong Choi
FADU Inc.

Eyee Hyun Nam
FADU Inc.

Eunji Lee
Soongsil University

Sungjin Lee
DGIST

Bryan S. Kim
Syracuse University

Abstract

This paper investigates the feasibility of using inexpensive flash memory on new interconnect technologies such as CXL (Compute Express Link) to overcome the memory wall. We explore the design space of a CXL-enabled flash device and show that techniques such as caching and prefetching can help mitigate the concerns regarding flash memory's performance and lifetime. We demonstrate using real-world application traces that these techniques enable the CXL device to have an estimated lifetime of at least 3.1 years and serve 68–91% of the memory requests under a microsecond. We analyze the limitations of existing techniques and suggest system-level changes to achieve a DRAM-level performance using flash.

1 Introduction

The growing imbalance between computing power and memory capacity requirement in computing systems has developed into a challenge known as the memory wall [23, 34, 52]. Figure 1, based on the data from Gholami et al. [34] and expanded with more recent data [11, 30, 43], illustrates the rapid growth in NLP (natural language processing) models (14.1× per year), which far outpaces that of memory capacity (1.3× per year). The memory wall forces modern data-intensive applications such as databases [8, 10, 14, 20], data analytics [1, 35], and machine learning (ML) [45, 48, 66] to either be aware of their memory usage [61] or implement user-level memory management [66] to avoid expensive page swaps [37, 53]. As a result, overcoming the memory wall in an application-transparent manner is an active research avenue; approaches such as creating an ML-centric system [45, 48, 61], building a memory disaggregation framework [36, 37, 52, 69], and designing new memory architecture [23, 42] are actively pursued.

We question whether it is possible to overcome the memory wall using flash memory — a memory technology that is typically used in storage due to its high density and capacity scaling [59]. While DRAM can only scale to gigabytes in capacity, a flash memory-based solid-state drive (SSD) is

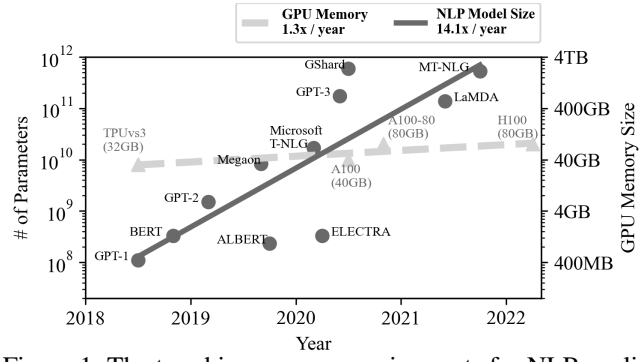


Figure 1: The trend in memory requirements for NLP applications [11, 30, 34, 43]. The number of parameters increases by a factor of 14.1× per year, while the memory capacity in GPUs only grows by a factor of 1.3× every year.

in the terabyte scale [23], a sufficiently large capacity to address the memory wall challenge. The use of flash memory as main memory is enabled by the recent emergence of interconnect technologies such as CXL [3], Gen-Z [7], CCIX [2], and OpenCAPI [12], which allow PCIe (Peripheral Component Interconnect Express) devices to be accessed directly by the CPU through load/store instructions. Furthermore, these technologies promise excellent scalability as more PCIe devices can be attached across switches [13] unlike DIMM (Dual Inline Memory Module) used for DRAM.

However, there are three main challenges to using flash memory as CPU-accessible main memory. First, there is a granularity mismatch between memory requests and flash memory. This results in a significant traffic amplification on top of the existing need for indirection in flash [23, 33]: for example, a 64B cache line flush to the CXL-enabled flash would result in 16KiB flash memory page read, 64B update, and 16KiB flash program to a different location (assuming a 16KiB page-level mapping). Second, flash memory is still orders of magnitude slower than DRAM (tens of microseconds vs. tens of nanoseconds) [5, 24]. As a consequence, while the peak data transfer rate between the two technologies is similar [4, 15], the long flash memory latency hinders sustained performance as data-intensive applications can only endure

latency within the microsecond range at most [53]. Lastly, flash memory has limited endurance and wears out after repeated writes [24, 44]. This limits the usability of the memory technology as flash memory blocks beyond their endurance limit exhibit unreliable behavior and high levels of errors [44].

We address the above flash memory challenges by exploring design options, particularly those related to caching and prefetching, so that a CXL-enabled flash device (or CXL-flash) can be used to overcome the memory wall. Even though prior works have explored the scalability aspects of multiple CXL devices [36, 42] and have proven the feasibility of CXL-flash [9, 42], to the best of our knowledge, we are the first open-sourced in-depth study on the design choices of a CXL-flash device and on the effectiveness of existing optimization techniques. Due to the large design space, we first explore the CXL-flash hardware design in § 4 and then evaluate and analyze detailed policies and algorithms in § 5. We discover that it is possible to design a CXL-flash with 68–91% of its requests achieving less than a microsecond latency and an estimated lifetime of at least 3.1 years using memory traces of real applications. While exploring various designs and policies, we make seven observations which collectively indicate that modern prefetching algorithms are ill-suited to predict memory access patterns for the CXL-flash. More specifically, the virtual to physical address translation obfuscates access patterns for existing prefetchers to perform adequately. To counter this, we explore passing memory access hints from the kernel to the CXL-flash to further improve performance.

We make the following contributions with this work.

- We develop a novel tool that collects physical memory traces of an application, and we simulate the behavior of a CXL-flash with these traces. Both the memory tracing tool and the CXL-flash simulator (§ 3) are available at https://github.com/spypaul/MQSim_CXL.git
- Through synthetic workloads, we demonstrate the potential to effectively reduce the latency of a CXL-flash by integrating various system design techniques such as caching and prefetching, highlighting optimization opportunities. (§ 4)
- Using real-world workloads, we analyze the limitations of the current prefetchers and suggest system-level changes for future CXL-flash to achieve near-DRAM performance, specifically sub-µs latencies for the device. (§ 5)

2 Background

In this section, we first describe the opportunities presented by CXL (Compute Express Link) [3] as a representation of PCIe-based memory coherent interconnect technologies (which also include Gen-Z [7], CCIX [2], and OpenCAPI [12]). We then discuss the challenges of using flash memory with CXL.

2.1 Opportunities presented by CXL

CXL is a new interconnect protocol built on top of PCIe that integrates CPUs, accelerators, and memory devices into a single computing domain [42]. The main benefits of this integration are twofold. First, it allows coherent memory access between CPUs and PCIe devices. This reduces the synchronization overheads that are typically required for data transfers between the CPU and the device. Second, it is easy to scale the number of CXL devices: through a CXL switch, another set of CXL devices can be connected to the CPU.

Among the three types of devices that CXL supports, the Type 3 device for memory expansion is of interest to this work. Type 3 devices expose host-managed device memory (HDM), and the CXL protocol allows the host CPU to directly manipulate the device memory via load/store instructions [3]. While CXL currently only considers DRAM and PMEM as the primary memory expansion devices, it is possible to use SSDs, thanks to CXL's coherent memory access [42]. Moreover, the high capacity and better scaling of flash-based SSDs, enabled by stacking in 3D [59] and storing multiple bits in a cell [24], can effectively address the memory wall that modern data-intensive applications face. Inspired by previous works on CXL [36, 42], this paper studies the feasibility of using flash memory as a CXL memory expansion device.

2.2 Challenges with flash memory

We discuss the following three peculiarities of flash that make it challenging to use it as the system's main memory.

Granularity mismatch. Flash memory is not randomly accessible: its data are written and read at page granularity whose size is in the order of kilobytes [33], resulting in a large traffic amplification. Furthermore, a page cannot be overwritten. Instead, a block, which consists of hundreds of pages, must be erased first, and data can be written to only erased pages [33]. This restrictive interface causes any 64B cache line flush to incur a large write amplification through read-modify-writes. An SSD, as a block device whose access granularity is much larger (4KiB), has far less overhead.

Microsecond-level latency. Flash memory is orders of magnitude slower than DRAM, whose latencies are in the range of tens to hundreds of nanoseconds. The relatively faster flash memory read is still in the tens of microseconds, while the slower program and erase operations are in the hundreds and thousands of microseconds. Moreover, the flash memory latency also depends on their cell technology [24]. As outlined in Table 1 as an example, as more bits are stored per cell, the latency increases, from SLC (single-level cell) to TLC (triple-level cell). The ultra-low latency (ULL) flash is a variant of SLC that improves performance at the expense of density [46, 76]. Even the ULL technology, however, is orders of magnitude slower than DRAM. As a block device, microsecond-level latencies are tolerable due to the software

Table 1: Overview of memory technology characteristics.

Technology	Read latency	Program latency	Erase latency	Endurance limit
DRAM [50]	46ns	46ns	N/A	N/A
ULL [46, 76]	3μs	100μs	1000μs	100K
SLC [24]	25μs	200μs	1500μs	100K
MLC [24]	50μs	600μs	3000μs	10K
TLC [24]	75μs	900μs	4500μs	3K

overhead in the storage stack. However, for a memory device that is directly accessed using load/store instructions, microsecond-level latencies become a challenge.

Limited endurance. The high voltages applied to flash memory during the program and erase operations slowly wear out the cells, making them unusable over time [44, 72]. The memory manufacturers specify an endurance limit as a guide to how many times the flash memory block can be erased. This limit also depends on the flash technology, as shown in Table 1. While this is nevertheless a soft limit and flash memory can still be used beyond the limit [72], worn-out blocks exhibit unreliable behavior and are not guaranteed to correctly store data [44]. Due to application-level and kernel-level caching and buffering, the amount of writes to a block-interfaced SSD is reduced, and the current endurance limit is often sufficient during the SSD's lifetime. As a memory device, however, flash memory will quickly become unusable with frequent memory writes.

We note that while these challenges for flash also exist in the storage domain, they are handled by the SSD-internal firmware. For CXL-flash, however, they should be addressed by hardware due to the much finer timescale, making it difficult to implement flexible and optimal algorithms. Thus, we expect these challenges to exacerbate when moving flash memory from the storage domain to the memory domain.

3 Tool and Methodology

To understand the behavior of physical memory accesses from the CPU to the CXL device, we build a physical memory tracing tool using page fault events (§ 3.1). We then demonstrate the necessity of this tool by comparing it with a set of virtual memory traces (§ 3.2). The tools and artifacts generated in this work are publicly available.

3.1 Tracing memory accesses

Main memory and CXL-flash are accessed via physical memory addresses. Unfortunately, to the best of our knowledge, no publicly available tool traces the physical memory transactions between the last level cache (LLC) and the memory controller without hardware modifications. Tracing the load/store instructions in the CPU is not sufficient as (1) it collects the virtual address accesses, and (2) the eventual accesses to the CXL-flash are filtered by the cache hierarchy.

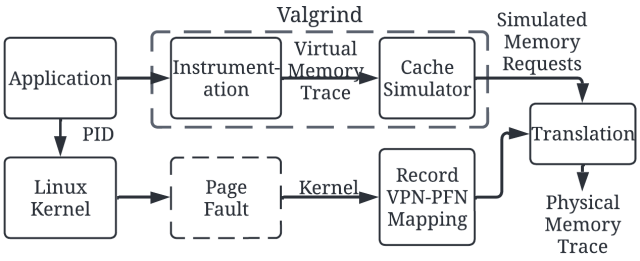


Figure 2: Workflow for collecting physical memory traces. We collect the virtual memory trace using Valgrind and simulate its behavior in the cache. Simultaneously, we capture page fault events to trace the updates to the page table, and use this to generate physical memory traces.

Table 2: Synthetic workload characteristics.

Workload	Inter-arrival time (ns)	Read-write ratio	Footprint (GiB)
Hash map	329	53:47	<1
Matrix multiply	38	55:45	<1
Min heap	72	50:50	1
Random	76	50:50	4
Stride	146	50:50	8

We trace physical memory accesses by combining memory tracing from Valgrind [19, 57] and information from page fault events. Figure 2 illustrates this workflow. As shown in the top path, we instrument the application with Valgrind for load/store instructions and use its cache simulator (Cachegrind) to filter accesses to memory. More specifically, we modify Cachegrind to collect memory accesses caused by LLC misses or evictions. However, these memory accesses from Cachegrind are still addressed virtually, and the virtual-to-physical (V2P) mapping information is needed to generate the physical memory trace. For this, as shown in the bottom path of Figure 2, we collect updates to the page table caused by page faults while the application is running. We modify kernel functions that install page table entries (`do_anonymous_page()` and `do_set_pte()`) and store the V2P translations for the target application's PID in the `/proc` file system. This captures the dynamic nature of page table updates during the execution of the application with minimal overhead. We combine the virtual accesses from Valgrind and the page table updates to generate the physical memory trace.

3.2 Virtual vs. physical memory accesses

We demonstrate our physical memory tracing tool using five synthetic applications based on prior work on prefetching [25, 56]. The characteristics of collected traces are summarized in Table 2. We collect the first 20 million memory accesses:

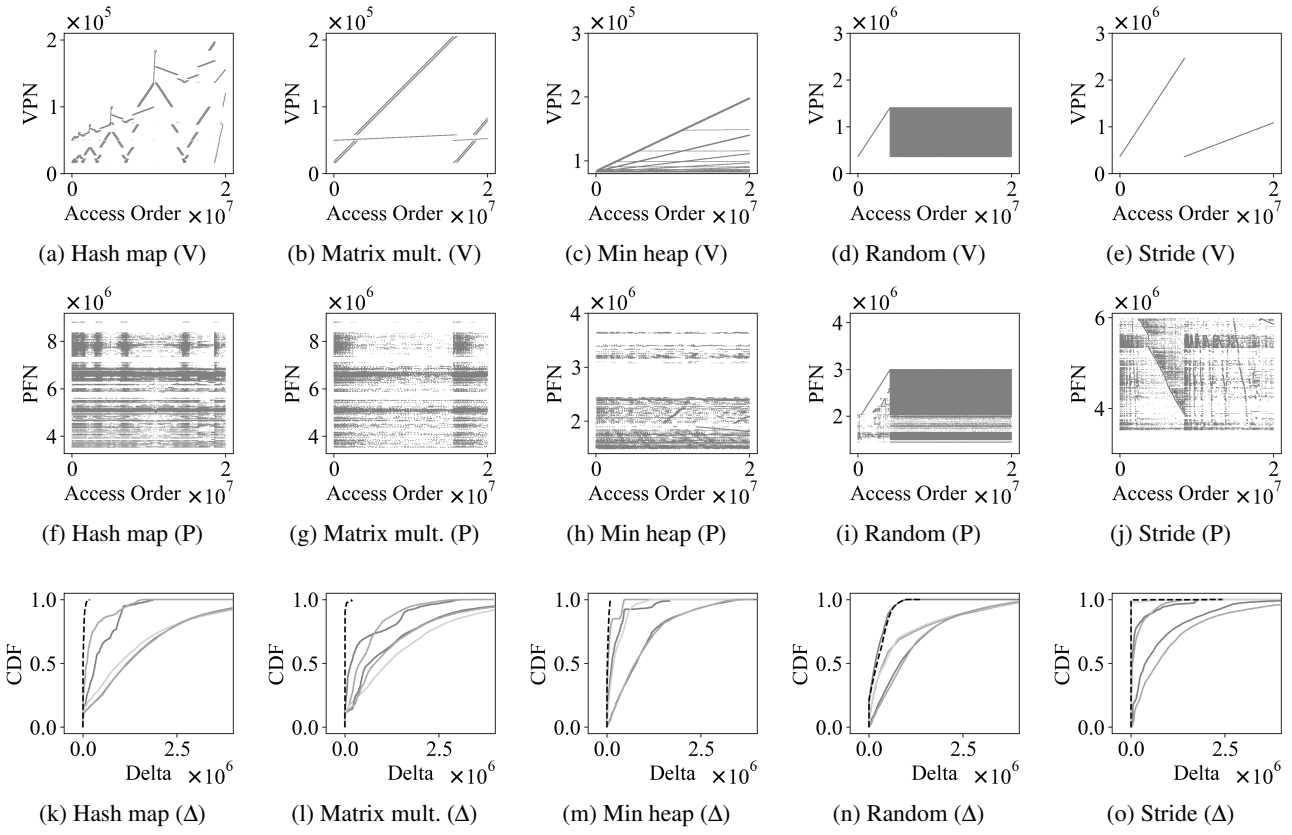


Figure 3: The scatter plots showing access patterns for five synthetic applications: hash map, matrix multiply, min heap, random, and stride. The top row (Figures 3a–3e) shows the virtual address accesses; the second row (Figures 3f–3j), the physical accesses. The last row (Figures 3k–3o) shows the CDF of the difference between consecutive accesses. We observe that the physical memory accesses appear different from the virtual ones due to address translation.

note these are not load/store instructions, but the memory transactions between the LLC and memory.

Figures 3a–3e (first row of Figure 3) plot the accessed virtual page number (VPN) for the five synthetic traces. We can observe that the virtual address access pattern matches our expectations for the application. However, as shown in Figures 3f–3j (second row of Figure 3), the corresponding physical frame number (PFN) does not resemble the VPN. We show the difference between consecutive accesses (Δ, delta) in Figures 3k–3o (last row of Figure 3). The black dashed line is the delta for the virtual address while the grey solid lines are the deltas for the physical addresses across five iterations, with the two of the iterations running while other applications are running to inflate the memory utilization. We make two observations. First, the virtual access patterns (black dashed lines) have much smaller delta values on average. However, the physical access patterns (grey solid lines) may have very large delta values due to virtual-to-physical address translation. Second, the grey solid lines rarely overlap with each other, highlighting that the physical memory pattern is dynamic and depends on various runtime factors that affect memory allocation. To this end, the observed mismatch between the physical and virtual addresses may be influenced by dynamic factors, such as memory utilization of the system.

To demonstrate that it is necessary to capture the physical memory trace, we measure the performance of a CXL-flash using the virtual and physical address traces as inputs. The CXL-flash is configured to have a flash memory backend of 8 channels and 8 ways per channel and a 512MiB DRAM cache, and implements a Next-N-line prefetcher [41] (more details in § 4). We measure the percentage of memory requests with less than a microsecond latency for the five synthetic applications and report the results in Table 3. Using virtual memory traces generates an *overly optimistic* result with far more requests completing under a microsecond compared to the result from running physical memory traces. The error between the running with virtual address and physical address is significantly high: all the matrix multiply experiments have errors over 25%. The random and stride access workload have low error rates, making it either too difficult or too easy to predict access patterns regardless of virtual or physical addressing.

One technique to mitigate the change of information during

Table 3: Percentage of sub-μs latencies for a CXL-flash using virtual and physical address traces for the five synthetic applications. We repeat the physical trace generation five times with iterations 4 and 5 having a higher system memory utilization (thus, a more fragmented memory layout). We compute the errors for the virtual trace performances in relation to those of the physical traces, and highlight errors over 10% in yellow (), and over 25% in red ().

Workload	% of sub-μs latency (virtual)	% of sub-μs latency (physical)					Error (%)				
		1	2	3	4	5	1	2	3	4	5
Hash map	96.9%	86.7%	88.3%	74.5%	63.8%	63.9%	10.2%	8.6%	22.4%	33.1%	33.0%
Matrix mult.	98.2%	72.7%	57.4%	59.2%	48.1%	47.9%	25.5%	40.8%	39.0%	50.1%	50.3%
Min heap	97.8%	92.1%	96.0%	75.6%	69.1%	69.4%	5.7%	1.8%	22.2%	28.7%	28.4%
Random	32.2%	26.4%	27.1%	28.0%	22.4%	21.8%	5.8%	5.1%	4.2%	9.8%	10.6%
Stride	64.7%	64.3%	59.4%	64.5%	51.9%	52.0%	0.4%	5.3%	0.2%	12.6%	12.7%

address translation is to utilize huge pages, which can significantly reduce the number of address translations [54, 58] to preserve memory access patterns. However, such a method can only reduce its impact on the system partially, and the address translation is inevitable. With the rapid growth of memory requirements of applications (14.1 × per year from Figure 1), within a few years, huge pages would exhibit the same challenges that the smaller pages face. Therefore, we decide to keep the configuration general to explore the design options for a CXL-flash.

4 Design Space for CXL-flash

We explore the design space for building a CXL-flash, specifically on the hardware modules inside it; we later evaluate algorithms and policies in § 5. To model the hardware, we build a CXL-flash simulator based on MQSim [68] and its extension MQSim-E [49], and use the physical memory traces of the five synthetic applications (Table 2) to evaluate the effects of design options. The overall architecture of our CXL-flash is depicted in Figure 4 with starting configuration in Table 4.

We answer the following research questions in this section.

- How effective is caching in improving performance? (§ 4.1)
- How can we effectively reduce flash memory traffic? (§ 4.2)
- How effective is prefetching in hiding the long flash memory latency? (§ 4.3)
- What are the appropriate flash memory technology and parallelism for CXL-flash? (§ 4.4)

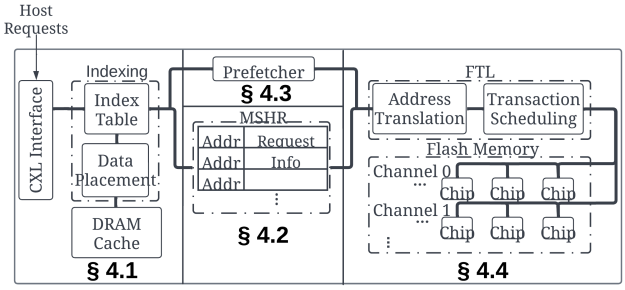


Figure 4: Architecture of the CXL-flash

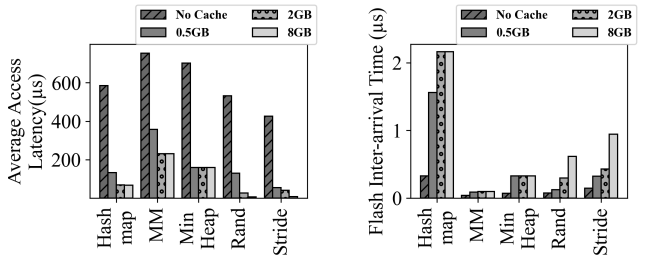
Table 4: Initial configuration for the CXL-flash in § 4.

Parameters	Value
DRAM latency	46ns
Cache replacement	FIFO
Flash parallelism	32 × 32
Flash technology	SLC (Table 1)

4.1 Caching for performance

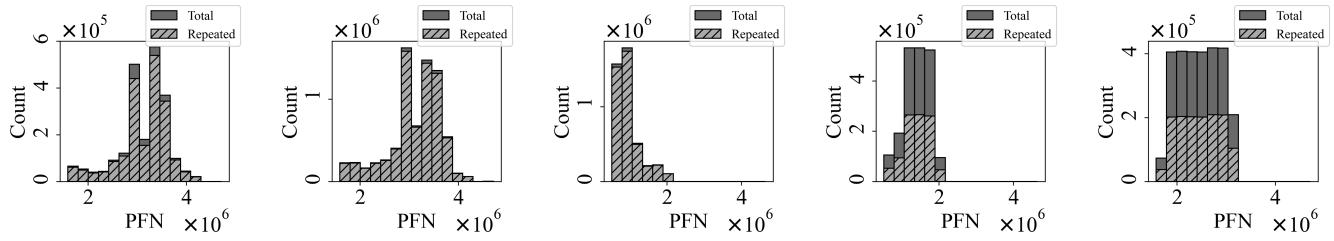
We first explore the effect of adding a DRAM cache in front of flash memory. The cache mainly serves two purposes. First, it improves the performance of the CXL-flash by serving frequently accessed data from the faster DRAM. Second, it reduces the overall traffic to flash memory on a cache hit.

Figure 5 quantitatively shows the benefit of using a cache. We vary the cache size from 0 to 8GiB and measure the average latency for the physical memory accesses (Figure 5a) and the inter-arrival time of flash memory requests to the backend. (Figure 5b). When there is no cache, the average latency is much higher than the flash memory read and program latencies because of queuing delays. This is even though the flash memory backend is configured to have an ample amount of



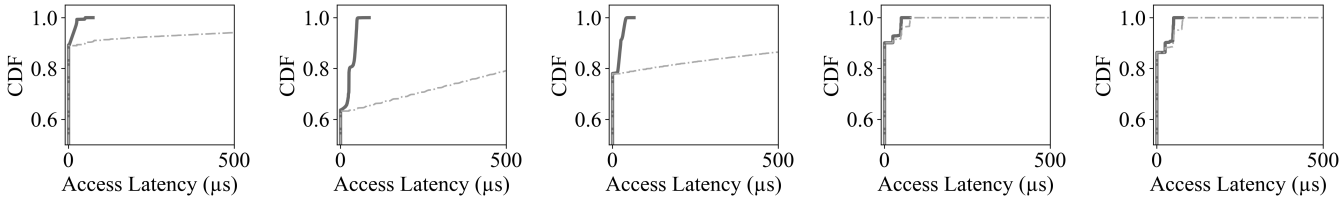
(a) Average access latency (b) Flash memory traffic

Figure 5: Average access latency (Figure 5a) for both 64B read and write requests and flash memory traffic (Figure 5b) as the DRAM cache size varies. In general, the cache improves performance and reduces the amount of traffic to flash. However, even with a sufficiently large cache, the average latencies are still much higher than that of DRAM due to the high intensity of memory accesses.



(a) Hash map (b) Matrix mult. (c) Min heap (d) Random (e) Stride

Figure 6: Flash memory read count for physical memory frames. The solid bar represents the total number of reads, while the shaded bar, the number of repeated reads. A repeated read is a read request to an outstanding read request.



(a) Hash map (b) Matrix mult. (c) Min heap (d) Random (e) Stride

Figure 7: The latency distribution with (solid lines) and without (dashed lines) MSHR.

parallelism at 32 channels and 32 ways per channel, it is insufficient to process the memory requests with short inter-arrival times. Adding a cache significantly reduces the traffic to the flash memory backend and improves the overall performance. However, we observe in Figure 5a that the average latencies for matrix multiply and min heap are still much higher than the DRAM latency, even though the memory footprints for these workloads are smaller than the cache. This is due to the short inter-arrival time of requests that overwhelm the flash memory backend for fetching data (Figure 5b). This experiment shows that caching alone is insufficient in reducing the latency of a CXL-flash, and we need additional auxiliary structures to reduce the traffic to flash memory.

4.2 Reducing flash memory traffic

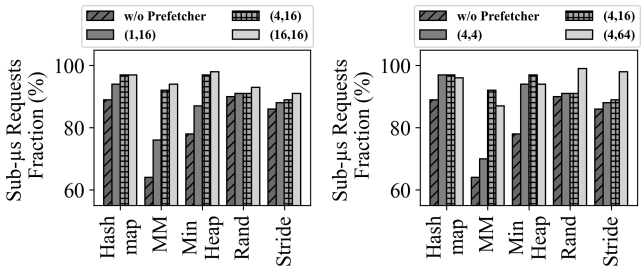
Memory accesses are at 64B granularity while the flash memory backend is addressed at 4KiB units. Thus, upon a cache miss, 4KiB of data will be fetched from flash, and subsequent 64B cache misses that belong to the same 4KiB will generate additional flash memory read requests even when the flash memory read is in progress. This scenario is very likely for memory accesses with high spatial locality, and exacerbated by the much longer flash memory latency. We call these *repeated reads*, and Figure 6 illustrates the severity of repeated reads in the hash map, matrix multiply, and heap workloads: over 90% of flash memory reads are repeats!

Inspired by CPU caches, we add a set of MSHRs (miss status holding registers) [29, 47] to the CXL-flash, as shown in Figure 4. MSHR tracks the current outstanding flash memory requests and services multiple 64B memory accesses from a single flash memory read. We note that MSHRs are uncommon in SSDs: in the storage domain, the software stack merges block I/Os with overlapping addresses so there is no need for the underlying device to implement MSHRs. However, for CXL-flash, there is no software layer to perform this duty as it receives memory transactions directly from the LLC. We observe in Figure 7 that MSHRs significantly reduce long tail latencies, particularly for the three workloads with a significant number of repeated reads. We also observe small improvements to the other two workloads, random and stride, by adding MSHR. However, MSHRs only reduce flash memory traffic, and it does not actively improve the cache hit rate by bringing data into the cache before they are needed.

4.3 Prefetching data from flash

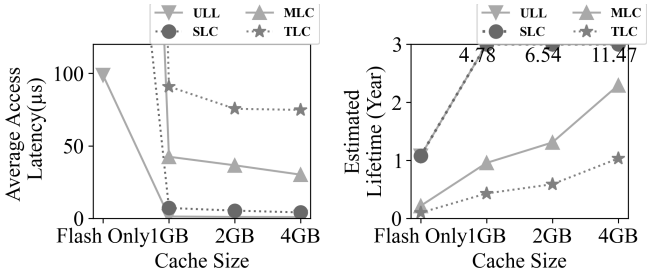
Prefetching is an effective technique for hiding the long latency of a slower technology. Typically prefetchers fetch additional data upon a demand miss or a prefetch hit. To understand the effectiveness of this technique, we implement a simple Next-N-line prefetcher [41] in our CXL-flash, as shown in Figure 4. This prefetcher has two configurable parameters: degree and offset. The degree controls the amount of additional data to fetch while the offset determines the prefetch address from the triggering one. In other words, the degree parameter represents the aggressiveness of the prefetcher, and the offset controls how far ahead the prefetcher is fetching.

Figure 8 shows the effect of varying the degree and offset for the prefetcher. For the (X, Y) notation in a unit of the



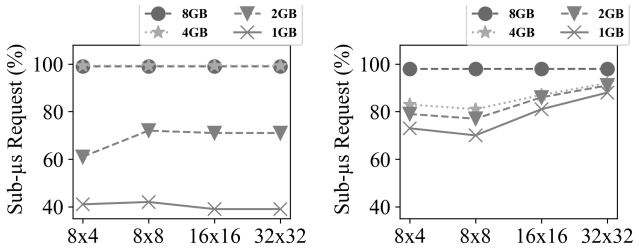
(a) Sensitivity to degree (b) Sensitivity to offset

Figure 8: The performance of the CXL-flash with different prefetcher configurations. (X, Y) represents the degree and offset for the Next-N-line prefetcher.



(a) Average access latency (b) Estimated lifetime

Figure 9: Sensitivity test to flash technology and cache size on the performance and lifetime of CXL-flash with stride.



(a) Random (b) Stride

Figure 10: Percentage of sub-μs requests when varying flash parallelism and cache size. The x-axis represents the flash memory parallelism (channels $\times$ ways). The lines represent values for different cache sizes.

number of 4KiB pages, X denotes the degree, and Y is the offset. As shown in Figure 8a, increasing the degree, or the aggressiveness of the prefetcher generally improves the performance. Even a small degree of 1 increases the portion of sub-microsecond requests from 64% to 76% for the matrix multiply workload, highlighting the necessity of prefetching for CXL-flash. However, the improvement plateaus and further increasing the degree may only pollute the cache. On the other hand, increasing the offset shows two different behaviors depending on the workload. For the hash map, matrix multiply, and min heap workloads, the performance first improves when increasing the offset from 4 to 16. However, an offset of 64 deteriorates the performance as it fetches data too far out. The random workload is insensitive to the offset unless it is large enough, while the stride workload shows gradual improvement as the offset increases.

4.4 Exploring flash technology and parallelism

In previous subsections, we examine the performance of the CXL-flash using SLC flash technology and ample flash parallelism of 32 channels and 32 ways per channel (32 $\times$ 32). In this section, we experiment with how sensitive technology (ULL, SLC, MLC, and TLC) and parallelism (8 $\times$ 4, 8 $\times$ 8, 16 $\times$ 16, and 32 $\times$ 32) are to the overall CXL-flash performance.

We first examine the effect of flash technology and cache size for the stride workload as shown in Figure 9. We use this workload as it performs well in the default configuration, thus we expect it to represent the workload with the lowest room for improvement. Figure 9a illustrates the average latency for the different memory technologies. We observe that even though ULL and SLC flash have a noticeable difference in latency (3μs vs. 25μs), the performance difference between the two is negligible with the existence of a cache. Only when there is no DRAM cache, ULL flash is outstandingly better. We also observe that using MLC and TLC technology degrades the performance significantly. Figure 9b shows the estimated lifetime of the CXL-flash based on the amount of flash write traffic. This estimation takes into consideration the endurance limit, the capacity, and the amount of data written; it optimistically assumes that the write amplification at the flash memory backend is negligible. We observe that with ULL and SLC technologies and some cache, the CXL-flash can achieve a lifetime of more than 4 years. Increasing the size of the cache further improves the lifetime due to reduced flash write traffic. For MLC- and TLC-based CXL-flash, it would only be viable with a sufficiently large cache: with only a 1GiB of cache, it would not last more than a year.

Next, we investigate the effect of varying flash parallelism and cache size on the overall performance using the random (Figure 10a) and stride (Figure 10b) workloads. We use these two as they have the largest memory footprint (8GiB and 4GiB, respectively). Flash technology here is SLC. We observe that with a sufficiently large cache, reducing parallelism to (8 $\times$ 4) does not adversely affect the performance. However, with smaller caches, the flash parallelism matters. Interestingly, the two workloads exhibit slightly different behaviors. The random workload shows high sensitivity to the cache size. On the other hand, the stride workload is less sensitive to the cache size but more to the parallelism. This is due to the prefetcher's effectiveness with stride workloads.

Table 5: Workload characteristics of real-world applications. The spatial and temporal locality values range between 0 and 1, and are computed using the stack and block affinity metrics [32]: a higher value indicates higher locality.

Workload	Category	Description	Inter-arrival time (ns)	# of accesses (M)	Read-write ratio	Footprint (GiB)	Spatial locality	Temporal locality
BERT [18]	NLP	Infers using a transformer model	297	41	73:27	0.5	0.64	0.66
Page rank [6]	Graph	Computes the page rank score	51	435	68:32	3.7	0.40	0.42
Radiosity [17]	HPC	Computes the distribution of light	1863	61	84:16	1.6	0.93	0.87
XZ [21]	SPEC	Compresses data in memory	237	71	55:45	0.9	0.31	0.38
YCSB F [22]	KVS	Read-modify-writes on Redis [14]	1137	110	65:35	1.8	0.56	0.55

4.5 Summary of findings

We briefly summarize our findings.

- Caching alone is not sufficient to hide the much longer flash memory latencies (§ 4.1), and we need auxiliary structures to reduce the flash memory traffic (§ 4.2).
- Prefetching data improves the CXL-flash's performance, but the best configuration (or even the algorithm) is workload-dependent (§ 4.3).
- The performance difference between using ULL and SLC is only marginal, and it is challenging to utilize MLC and TLC flash in terms of both performance and lifetime (§ 4.4).

5 Evaluation of Policies

Building on top of our design space exploration for the CXL-flash architecture from § 4, we evaluate advanced caching and prefetching policies in this section. We use five different real-world applications that are memory-intensive from a wide variety of domains: natural language processing [18, 70], graph processing [6, 27], high-performance computing [17, 62], SPEC CPU [16, 21], and key-value store [22, 31]. We collect the physical memory traces using our tool (§ 3) and summarize the workload characteristics in Table 5.

However, the memory footprints of the real applications are smaller than we had anticipated, even though they are collected on a machine with 64GiB of memory. Thus, we intentionally configure the cache to be small (64MiB) so that experimental results would scale up for larger workloads. We also scale down the flash parallelism to a more realistic setting and use ULL flash. The default parameters for the CXL-flash in this section are summarized in Table 6.

Table 6: Default parameters for the CXL-flash in § 5.

Parameters	Value
DRAM size	64MiB
DRAM latency	46ns
Flash parallelism	8×8
Flash technology	ULL (Table 1)

5.1 Cache replacement policy

Unlike the previous examination of cache size on performance (§ 4.1), here we fix the cache size and evaluate the effects of different cache replacement policies across different set associativities. In particular, we implement the following four.

FIFO evicts the oldest data.

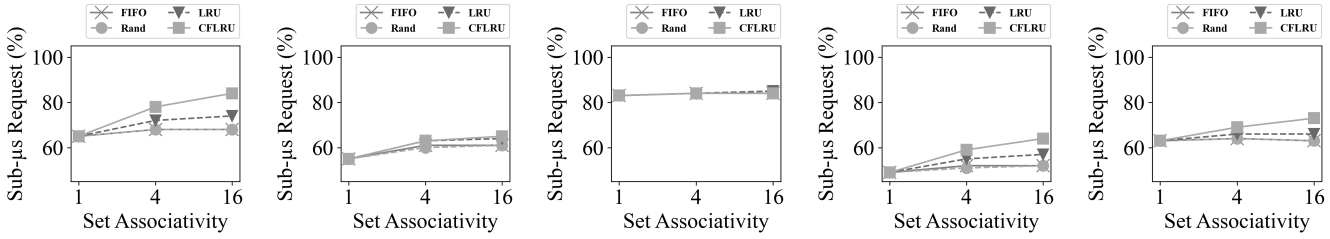
Random selects data arbitrarily to evict.

LRU kicks out the least recently used data.

CFLRU [60] prefers to evict clean data over modified ones.

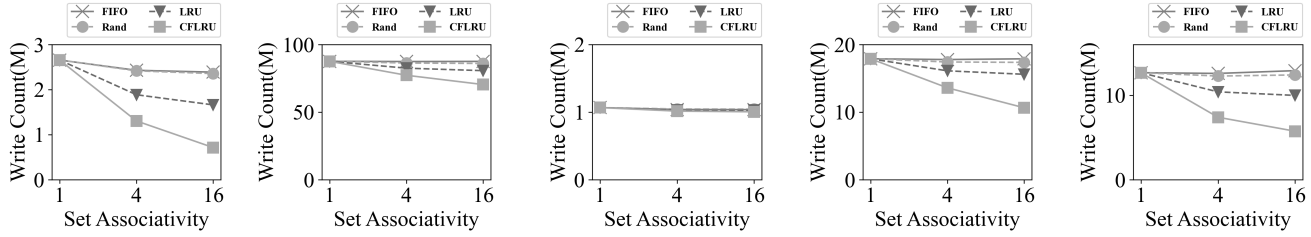
We select Random as our baseline, and FIFO and LRU are two standard CPU cache policies implementable in hardware. To further reduce traffic and extend the device's lifetime, we implement CFLRU to explore the benefits of prioritizing evicting clean cache lines to reduce flash write activities.

Figure 11 measures the percentage of memory requests to the CXL-flash with less than a microsecond latency, and Figure 12 shows the number of flash memory writes. We make five observations from these figures. First, increasing associativity improves performance as it increases the cache hit rate. For a caching system whose miss penalty is high, increasing the hit rate has a greater impact than reducing hit time. Second, CFLRU outperforms the other replacement policies, particularly in BERT, XZ, and YCSB (Figures 11a, 11d, and 11e). This is supported by the significant reduction in write traffic as shown in Figures 12a, 12d, and 12e. Third, workloads with high localities such as Radiosity are insensitive to cache replacement policies (Figures 11c and 12c): at least 83% of the request have sub-microsecond latency regardless of the policy. Four, read-dominant workloads generally perform better than write-heavy ones as the flash memory program latency is disproportionately larger than that of read. BERT and Radiosity only generate as low as 0.7M and 1.0M flash writes, respectively (Figures 12a and 12c), and in turn, their portion of sub-microsecond latencies are as high as 84% and 85%, respectively (Figures 11a and 11c). Lastly, workloads with low localities not only perform poorly but also are less sensitive to the cache policies. In particular, as shown in Figure 11b, only at most 65% of the requests achieve a sub-microsecond latency for the page rank workload due to its low localities and large footprint. The XZ trace in Figure 11d



(a) BERT (b) Page Rank (c) Radiosity (d) XZ (e) YCSB

Figure 11: Percentage of CXL-flash latencies smaller than a microsecond with respect to cache replacement policies and set associativity. In general, increasing associativity reduces the latency and CFLRU performs better than the others.



(a) BERT (b) Page Rank (c) Radiosity (d) XZ (e) YCSB

Figure 12: Number of flash memory write requests with respect to cache replacement policies and set associativity. CFLRU noticeably reduces the number of writes as the associativity increases.

also exhibits low localities but is more sensitive to CFLRU than page rank as the workload has a higher write ratio.

In the storage domain, reducing the amount of data written to the SSD is achieved by various software-level techniques, including the OS-level page cache. Cache management for CXL-flash, however, behaves similarly to CPU caches, and there may be limitations to how close it approaches optimality.

5.2 Prefetching policy

Previously in § 4.3, we measured the effectiveness of a simple Next-N-line prefetcher with a large 8GiB cache. In this section, we scale down the cache to 64MiB, set its associativity to 16, and manage it using the CFLRU algorithm, and measure the performance of the following five prefetcher settings.

NP (No prefetch) does not prefetch any data.

NL (Next-N-line) [41] brings in the next *N* data upon a demand miss or prefetch hit.

FD (Feedback-directed) [65] dynamically adjusts the aggressiveness of the prefetcher by tracking prefetcher accuracy, timeliness, and pollution.

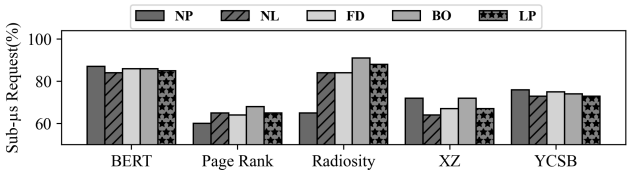
BO (Best-offset) [55] learns the deltas between consecutive accesses by tracking the history of recent requests. It also has a notion of confidence to disable prefetching.

LP (Leap) [53] implements a majority-based prefetching with dynamic window size adjustment. It also gradually adjusts aggressiveness based on the prefetcher accuracy.

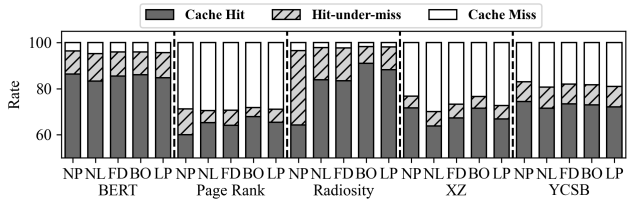
We select these algorithms as they are proven to be effective, implementable in hardware, and fit into the design space of a CXL-flash. In particular, NL, FD, and BO are prefetchers for CPU cache, where BO is an enhancement of NL, and FD utilizes performance metrics we will later discuss. LP is primarily for prefetching data from remote memory, where the latency difference between a cache hit and a cache miss can be similar to that in our design space.

Observation #1: ***Although 68–91% of requests have a latency of under a microsecond, using a prefetcher can be detrimental to the performance of real-world applications.*** As shown in Figure 13a, the state-of-the-art prefetchers degrade the performance for three workloads, BERT, XZ, and YCSB workloads, and are only helpful for the other two workloads. Radiosity, in particular, shows a 36% increase in sub-microsecond latencies when using the best-offset prefetcher. To understand why, we examine the cache hit, hit-under-miss (HUM), and miss rate in Figure 13b. A cache hit-under-miss refers to a hit in the MSHR where while the data is not in the cache yet, it is being fetched due to a previous miss. We observe that BO on Radiosity converts 25% of hit-under-miss into hits, indicating high effectiveness of prefetching on workloads with a high spatial locality factor (*cf.* Table 5). Our observation indicates that the performance of prefetchers depends on the characteristics of workloads, and they can have detrimental effects on applications.

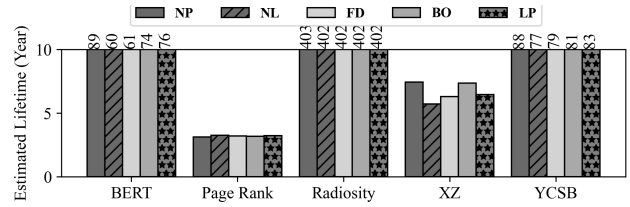
Observation #2: ***Even under high-intensity workloads, a CXL-flash has a lifetime of at least 3.1 years.*** We estimate the lifetime of the CXL-flash under real workloads based on the amount of data written to flash, endurance limit, and 1TiB capacity, as shown in Figure 13c. We observe, in the worst case, the device would last 3.1 years under Page Rank, but



(a) Percentage of sub-μs requests



(b) Cache hit, cache hit-under-miss (HUM), and cache miss



(c) Device lifetime

Figure 13: CXL-flash's performance and lifetime with different prefetchers. Figure 13a shows the portion of requests with a latency of less than a microsecond. Figure 13b shows the hit, hit-under-miss, and miss rate of the 64MiB cache inside the CXL-flash. Figure 13c plots the estimated lifetime.

under workloads such as Radiosity, it would be as much as 403 years. Three factors contribute to the lifetime: workload intensity, read-write ratio, and locality; Page rank has the highest workload intensity, a high ratio of writes, and a low locality. Even under this adverse condition, the CXL-flash provides a reasonable lifetime; hence, the durability of the CXL-flash can sustain the intensity of memory requests.

Observation #3: ***A CXL-flash has a better performance per cost than a DRAM-only device.*** While a CXL-flash falls slightly short of achieving a DRAM-like performance for sub-µs requests, our analysis reveals its potential to provide benefits for memory-intensive applications. As a CXL-flash can serve 68–91% of the memory requests under a microsecond, and the recent price point of DRAM is 17 - 100× higher than that of NAND flash [39, 64, 77], we expect an 11 - 91× performance-per-cost benefit from a CXL-flash over a DRAM-only device, as depicted in Figure 14. Although some cases may still prefer a DRAM-only device when achieving the best performance is essential, a CXL-flash can be a cost-effective memory expansion option depending on the workload.

Interestingly, we observe that while prefetchers are useful for Page Rank, their performance is overall the worst, with only at most 68% of requests completing under a microsecond. To further understand the performance of prefetchers, we measure the following four metrics.

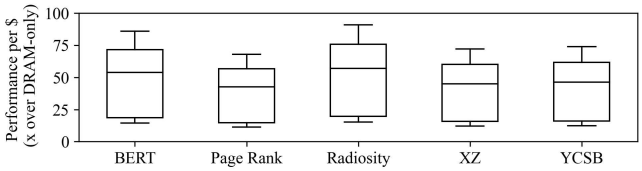


Figure 14: Performance-per-cost benefits of a CXL-flash with BO prefetcher over a DRAM-only device. The estimation is derived from the performance results in Figure 13a, the recent price point of DRAM at 5 $/GB [77], and the price range of NAND flash varying from 0.05 to 0.30 $/GB [39, 64].

Accuracy measures how much of the data brought in by the prefetcher is actually used. Higher is better.

Coverage is the portion of prefetched data cache hits among the memory requests. A high coverage means that cache hits are thanks to the prefetcher, while a low coverage indicates that the prefetcher is not contributing.

Lateness is the portion of late prefetches among all the prefetches. A prefetched data is late if it is accessed while it is being prefetched. Lower is better.

Pollution measures how many cache misses are caused by the prefetcher among cache misses. Lower is better.

Observation #4: ***In cases where the prefetcher improves the performance, it is due to achieving high accuracy.*** We plot the four metrics for the evaluated prefetchers in Figure 15. Lateness and pollution are negative metrics (the lower the better), so we invert their bars so that higher is better for all metrics. We observe that the defining characteristic for the workloads where the prefetcher is helpful (Page Rank and Radiosity) is that the accuracy is high. For example, the Leap (LP) prefetcher attains 85% accuracy under Radiosity while only reaching 27% under BERT. Additionally, the Best-offset (BO) prefetcher achieves 48% accuracy under XZ; however, its limited coverage of 4% suggests that despite achieving relatively higher accuracy, the prefetcher is not actively fetching data to contribute to performance improvement.

We further analyze Page Rank to understand why prefetchers are able to reach relatively high accuracy even though the workload has the lowest locality (computed using the stack and block affinity metrics [32]). As Figure 16 shows, the Page Rank exhibits distinct phases in their workload. During the first phase, Page Rank loads graph information and exhibits high locality (Figure 16a). The best-offset prefetcher is also able to attain high coverage and accuracy (Figure 16b). However, in the second phase, Page Rank builds the graph, and the access pattern here has a very low locality. Consequently, the best-offset prefetcher becomes more inactive (low coverage) as its accuracy drops. During the last phase, Page Rank computes the score for each vertex. While its access locality is not high, the prefetcher performs well and most of the accesses hit in the cache. Note that while the pollution is bad, the cache miss rate is very low so its impact on performance

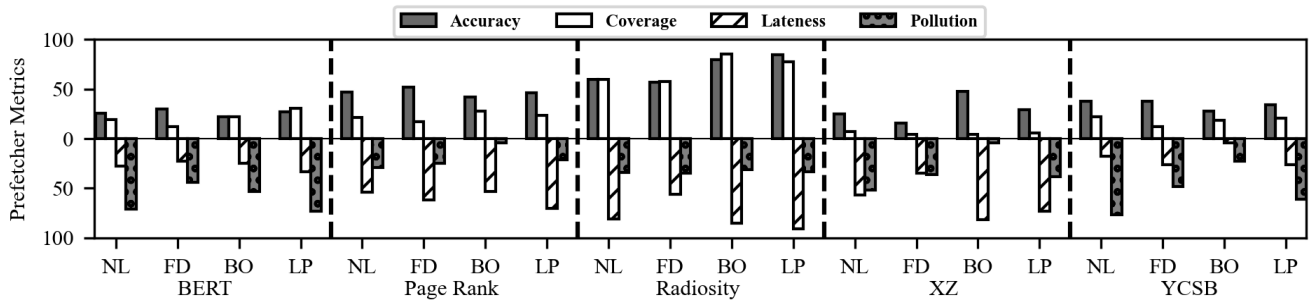
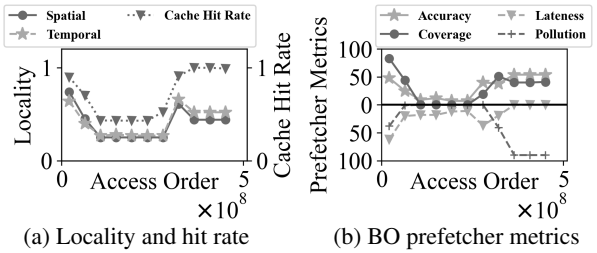


Figure 15: Accuracy, coverage, lateness, and pollution metrics for the prefetchers.



(a) Locality and hit rate (b) BO prefetcher metrics

Figure 16: Page Rank behavior over time.

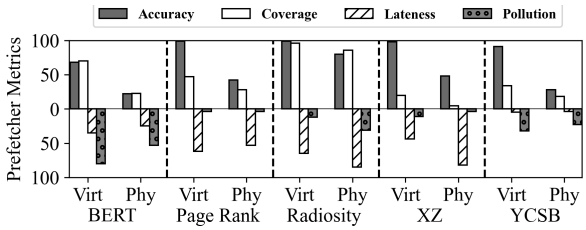
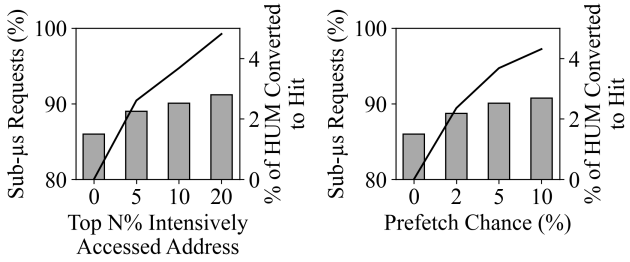


Figure 17: BO prefetcher metrics for virtual vs. physical.



(a) Sensitivity to # of addresses. (b) Sensitivity to hint chance.

Figure 18: Improvement in performance with memory access pattern hints for BERT. Figure 18a is the sensitivity to the number of addresses for which hints are provided. Figure 18b shows the performance improvement as more hints are added. In both figures, the line represents the number of hit-under-misses without hints converted to cache hits with hints.

is negligible. This analysis indicates that while the prefetcher is beneficial for the first and last phases, the low locality in the second phase limits the performance.

Observation #5: ***Cache pollution is the main reason behind performance degradation when the accuracy is low.*** As shown in Figure 15, BERT and YCSB have low accuracies while their pollutions are high, leading to a result where enabling prefetchers degrades performance (Figure 13a). For XZ, even though the accuracy of the best-offset (BO) prefetcher is low, it is no worse than no prefetcher as it causes little pollution. We attribute this to BO's ability to disable prefetching based on its accuracy. For Page Rank and Radiosity, prefetchers exhibit low pollution although their lateness is high. Cache pollution degrades the performance of a CXL-flash, and prefetchers should be aware of the impact to avoid having detrimental effects on the device.

Observation #6: ***The virtual-to-physical address translation makes it difficult for the CXL-flash to prefetch data.*** To understand the effect of V2P address translation, we simulate the CXL-flash with the best-offset prefetcher using the virtual memory traces of the five application workloads, and Figure 17 compares the four prefetcher metrics between virtual and physical traces. We make two observations. First, aside from Radiosity, we observe a significant drop in accuracy from virtual to physical traces. The BO prefetcher under Page Rank is 99% accurate for the virtual memory trace, but with the physical trace, accuracy drops to 42%. Second, coverage also drops, indicating that the prefetcher becomes less active under physical memory accesses: for example, it drops from 76% to 26% under BERT. The drop in both accuracy and coverage for physical traces shows that the CXL-flash would perform better if it were addressed virtually.

Observation #7: ***If the kernel were to provide memory access pattern hints to the device, the CXL-flash performance improves by converting hit-under-misses into cache hits.*** We consider a hypothetically clairvoyant kernel that knows the physical memory access pattern. This is not too far-fetched as data-intensive applications often iterate multiple times and their behaviors can be profiled. More specifically, we assume that the kernel has information on the top intensively accessed physical frames, and can pass hints to the device prior to their actual accesses. To limit the overhead of kernel involvement, we model a probabilistic generation of access hints. Figure 18a shows the performance improvement for BERT when hints are generated at 10% for the top N% of intensively accessed addresses. We observe that with access hints to more addresses, the percentage of sub-microsecond latencies increases from 86% to 91% by converting hit-under-

misses (HUM) into cache hits. Figure 18b considers a variable hint generation chance, from 0% to 10% for the top 10% of intensively accessed addresses. Similarly, we see an overall improvement in performance, though it plateaus at 91%. Our experiments show that host-generated access pattern hints leveraging the host's knowledge of the workload behaviors can potentially improve the CXL-flash performance.

6 Related Work

The evaluated cache policies and prefetching algorithms are well-studied in prior proposals. However, most of them are for managing and optimizing the CPU cache [41, 55, 65], where the latency difference between a cache hit and a cache miss is much smaller than that in a CXL-flash. CFLRU [60] and Leap [53] share a similar design space to our device; however, the memory access intensity they face is not as extreme as what a CXL-flash needs to handle. Therefore, it is crucial to evaluate the effectiveness of these policies and algorithms under the design space of a CXL-flash.

Techniques to mitigate the performance degradation due to address translations and limitations of flash has been explored in prior works. Utilizing huge pages can reduce the number of address translations [54, 58]. FlashMap [40] and FlatFlash [23] combine address translation of the SSD with the page table to reduce overheads. eNVY [73] employs write buffering, page remapping, and a cleaning policy to enable direct memory addressability and sustain performance. Future research in CXL host systems should further explore the potential benefits of host-generated hints and insights from these prior works to reduce the overheads.

Memory disaggregation organizes memory resources across servers as a network-attached memory pool, enabling meeting the high memory requirements for data-intensive applications [37, 38, 52, 69]. While our work does not directly investigate memory disaggregated systems, using CXL-flash as disaggregated memory helps overcome the memory wall.

Utilization of non-DRAM to expand memory has been explored in prior works [28,40,63,74]. HAMS [75] aggregates persistent memory and ULL flash as a memory expansion by managing data paths among hosts and memory hardware in an OS-transparent manner. Suzuki et al. [67] present a lightweight DMA-based interface that bypasses the NVMe protocols to enable flash read access with DRAM-like performance. SSDAlloc [26] is a memory manager and a runtime library that allows applications to use flash as a memory device through the OS paging mechanism, which can cause overheads when accessing data in SSDs. FlatFlash [23] exposes a flat memory space using DRAM and flash memory by integrating the OS paging mechanism and the SSD's internal mapping table. While these prior works primarily focus on OS-level management and host-device interaction, our work builds on top of them by investigating the design decisions internal to a memory expansion device.

Memory expansion with CXL Type 3 devices is an active research area [36, 42, 50, 71]. Pond [50] utilizes CXL to improve DRAM memory pooling in cloud environments and proposes machine-learning models to manage local and pooled memory. While this work investigates how to use a CXL Type 3 device in a cloud setting, our work studies how to implement one using flash memory. DirectCXL [36] successfully connects host processors with external DRAM via CXL in real hardware and develops a software runtime to directly access the resources. Lastly, CXL-SSD [42] advocates combining CXL and SSD to expand host memory. While we share the same goal with this work, it mainly discusses the CXL interconnect and scalability potentials of CXL-SSDs.

ML-specific designs build systems that address the memory wall challenge [45, 48, 51]. MC-DLA [48] proposes an architecture that aggregates memory modules to expand the memory capacity for training ML models on accelerators. Behemoth [45] observes that many NLP models require large amounts of memory but not a lot of bandwidth, and proposes a flash-centric training framework that manages data movement between memory and SSDs to overcome the memory wall.

7 Conclusion

We explore the design space of a CXL-flash device and evaluate existing optimization techniques. Using physical memory traces, we find that 68–91% of memory access achieves a sub-microsecond latency for a CXL-flash device that can last at least 3.1 years. We discover that the address translation for virtual memory makes it particularly difficult for the CXL-flash's prefetcher to be effective and suggest passing kernel-level access pattern hints to further improve the performance.

While we attempt to generalize the results by testing the device with a variety of workloads and design parameters, it is important to acknowledge a few limitations. The current design of a CXL-flash as explored in this paper does not consider the flash's internal tasks such as garbage collection and wear leveling. In addition, the host system considered may not fully reflect the new system characteristics introduced by CXL. Therefore, we believe more work needs to be done in the CXL-flash research space, and our work can be a platform on which future research can build upon.

Acknowledgements

We thank our Shepherd, Animesh Trivedi, and the anonymous reviewers for their insightful comments. We thank Jongmoo Choi for reviewing the early draft of this paper. This work is funded in part by FADU Inc., the National Research Foundation of Korea (NRF-2018R1A5A1060031), the MOTIE (Ministry of Trade, Industry & Energy) (1415181081), and the KSRC (Korea Semiconductor Research Consortium) (20019402).

References

[1] Apache Spark. https://spark.apache.org/.

[2] CCIX consortium. https://www.ccixconsortium.com/.

[3] CXL consortium. https://www.computeexpresslink.org/.

[4] DDR memory speeds and compatibility. https://www.crucial.com/support/memory-speeds-compatability.

[5] The difference between RAM speed and CAS latency. https://www.crucial.com/articles/about-memory/difference-between-speed-and-latency.

[6] GAP benchmark suite. https://github.com/sbeamer/gapbs.

[7] Gen-Z consortium. https://genzconsortium.org/specifications/.

[8] Memcached. http://memcached.org/.

[9] Memory-Semantic SSD. https://samsungmsl.com/ms-ssd/.

[10] MongoDB. https://www.mongodb.com/home.

[11] NVIDIA H100Tensor core GPU. https://www.nvidia.com/en-in/data-center/h100/.

[12] OpenCapi consortium. https://opencapi.org/.

[13] PCIe 5.0 multi-port switch. https://www.rambus.com/interface-ip/controllers/pci-express-controllers/pcie5-multi-port-switch/.

[14] Redis. https://redis.io/.

[15] Samsung PM9A3. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[16] SPEC CPU 2017. https://www.spec.org/cpu2017/.

[17] The Splash-3 benchmark suite. https://github.com/SakalisC/Splash-3.

[18] TensorFlow code and pre-trained models for BERT. https://github.com/google-research/bert.

[19] Valgrind. https://valgrind.org/.

[20] VoltDB. https://github.com/VoltDB/voltdb.

[21] XZ utils. https://www.tukaani.org/xz/.

[22] Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[23] Ahmed Abulila, Vikram Sharma 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 971–985. Association for Computing Machinery, 2019. https://doi.org/10.1145/3297858.3304061.

[24] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[25] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 513–526. Association for Computing Machinery, 2020. https://dl.acm.org/doi/10.1145/3373376.3378498.

[26] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 211–224. USENIX Association, 2011. https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Badam.pdf.

[27] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv:1508.03619 [cs.DC]*, 2015. https://arxiv.org/abs/1508.03619.

[28] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 217–228. Association for Computing Machinery, 2009. https://dl.acm.org/doi/10.1145/2528521.1508270.

[29] Xi E. Chen and Tor M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 59–70, 2008.

[30] Eli Collins and Zoubin Ghahramani. LaMDA: our breakthrough conversation technology, 2021. https://blog.google/technology/ai/lamda/.

[31] 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 '10, page 143–154. Association for Computing Machinery, 2010. https://dl.acm.org/doi/10.1145/1807128.1807152.

[32] Cory Fox, Dragan Lojpur, and An-I Andy Wang. Quantifying temporal and spatial localities in storage workloads and transformations by data path components. In *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, pages 1–10, 2008. https://ieeexplore.ieee.org/document/4770561.

[33] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005. https://dl.acm.org/doi/10.1145/1089733.1089735.

[34] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *RiseLab Medium Post*, 2021. https://github.com/amirgholami/ai_and_memory_wall.

[35] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613. USENIX Association, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez.

[36] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, high-performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294. USENIX Association, 2022. https://www.usenix.org/conference/atc22/presentation/gouk.

[37] 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. USENIX Association, 2017. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[38] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433. Association for Computing Machinery, 2022. https://dl.acm.org/doi/abs/10.1145/3503222.3507762.

[39] Gertjan Hemink and Akira Goda. 5 - nand flash technology status and perspectives. In Andrea Redaelli and Fabio Pellizzer, editors, *Semiconductor Memories and Systems*, Woodhead Publishing Series in Electronic and Optical Materials, pages 119–158. Woodhead Publishing, 2022. https://www.sciencedirect.com/science/article/pii/B9780128207581000030.

[40] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 580–591. Association for Computing Machinery, 2015. https://www.usenix.org/conference/hotstorage18/presentation/koh.

[41] Norman Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990. https://ieeexplore.ieee.org/document/134547.

[42] Myoungsoo Jung. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *HotStorage '22*, page 45–51. Association for Computing Machinery, 2022. https://doi.org/10.1145/3538643.3539745.

[43] Paresh Kharya and Ali Alvi. Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, the world's largest and most powerful generative language model, 2021. https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/.

[44] Bryan S. Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for SSD reliability. FAST'19, page 281–294. USENIX Association, 2019. https://www.usenix.org/conference/fast19/presentation/kim-bryan.

[45] Shine Kim, Yunho Jin, Gina Sohn, Jonghyun Bae, Tae Jun Ham, and Jae W. Lee. Behemoth: A flash-centric training accelerator for extreme-scale DNNs. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 371–385. USENIX Association, 2021. https://www.usenix.org/conference/fast21/presentation/kim.

[46] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of ultra-low latency solid state drives. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'18, page 15. USENIX Association, 2018. https://www.usenix.org/conference/hotstorage18/presentation/koh.

[47] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture, Minneapolis, MN, USA, May 1981*, pages 81–88, 1981. http://dl.acm.org/citation.cfm?id=801868.

[48] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric HPC system for deep learning. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, page 148–161. IEEE Press, 2018. https://dl.acm.org/doi/10.1109/MICRO.2018.00021.

[49] Dusol Lee, Duwon Hong, Wonil Choi, and Jihong Kim. MQSim-E: An enterprise SSD simulator. *IEEE Computer Architecture Letters*, 21(1):13–16, 2022.

[50] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2023. https://arxiv.org/abs/2203.00241.

[51] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 395–410. USENIX Association, 2019. https://dl.acm.org/doi/10.5555/3358807.3358841.

[52] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, page 267–278. Association for Computing Machinery, 2009. https://doi.org/10.1145/1555754.1555789.

[53] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, 2020. https://www.usenix.org/conference/atc20/presentation/al-maruf.

[54] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: Overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 121–131. Association for Computing Machinery, 2019. https://doi.org/10.1145/3319647.3325839.

[55] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480, 2016. https://ieeexplore.ieee.org/document/7446087.

[56] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2), 2016. https://dl.acm.org/doi/10.1145/2907071.

[57] Nicholas Nethercote and Julian 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. Association for Computing Machinery, 2007. https://doi.org/10.1145/1250734.1250746.

[58] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 679–692. Association for Computing Machinery, 2018. https://doi.org/10.1145/3173162.3173203.

[59] K. Parat. and A. Goda. Scaling trends in NAND flash. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 2.1.1–2.1.4, 2018. https://ieeexplore.ieee.org/document/8614694.

[60] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jinsoo Kim, and Joonwon Lee. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, page 234–241. Association for Computing Machinery, 2006. https://dl.acm.org/doi/10.1145/1176760.1176789.

[61] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. VDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1 – 13. IEEE Press, 2016.

[62] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, 2016. https://ieeexplore.ieee.org/abstract/document/7482078.

[63] Mohit Saxena and Michael M. Swift. FlashVM: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 14. USENIX Association, 2010. https://www.usenix.org/conference/usenix-atc-10/flashvm-virtual-memory-management-flash.

[64] Carol Sliwa. SSD and NAND flash prices will decline through start of 2021, 2020. https://www.techtarget.com/searchstorage/news/252487918/SSD-and-NAND-flash-prices-will-decline-through-start-of-2021.

[65] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, 2007. https://ieeexplore.ieee.org/document/4147648.

[66] Suhas Jayaram Subramanya, Harsha Vardhan Simhadri, Srajan Garg, Anil Kag, and Venkatesh Balasubramanian. BLAS-on-flash: An efficient alternative for large scale ML training and inference. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 469–484. USENIX Association, 2019. https://www.usenix.org/conference/nsdi19/presentation/subramanya.

[67] Tomoya Suzuki, Kazuhiro Hiwada, Hirotsugu Kajihara, Shintaro Sano, Shuou Nomura, and Tatsuo Shiozawa. Approaching dram performance by using microsecond-latency flash memory for small-sized random read accesses: A new access method and its graph applications. *Proc. VLDB Endow.*, 14(8):1311–1324, apr 2021. https://doi.org/10.14778/3457390.3457397.

[68] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQsim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, page 49–65. USENIX Association, 2018. https://www.usenix.org/conference/fast18/presentation/tavakkol.

[69] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, 2020. https://www.usenix.org/conference/atc20/presentation/tsai.

[70] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962v2*, 2019.

[71] Daniel Waddington, Moshik Hershcovitch, Swaminathan Sundararaman, and Clem Dickey. A case for using cache line deltas for high frequency VM snapshotting. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 526–539. Association for Computing Machinery, 2022. https://dl.acm.org/doi/abs/10.1145/3542929.3563481.

[72] Ellis Herbert Wilson, Myoungsoo Jung, and Mahmut T. Kandemir. ZombieNAND: Resurrecting dead NAND flash for improved SSD longevity. In *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 229–238, 2014. https://doi.org/10.1109/MASCOTS.2014.37.

[73] Michael Wu and Willy Zwaenepoel. Envy: A non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, page 86–97. Association for Computing Machinery, 1994. https://doi.org/10.1145/195473.195506.

[74] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven 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*, page 169–182. USENIX Association, 2020. https://www.usenix.org/conference/fast20/presentation/yang.

[75] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Nam Sung Kim, Mahmut Taylan Kandemir, and Myoungsoo Jung. Revamping storage class memory with hardware automated memory-over-storage solution. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 762–775. IEEE Press, 2021. https://dl.acm.org/doi/abs/10.1109/ISCA52012.2021.00065.

[76] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun

Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 477–492. USENIX Association, 2018. https://www.usenix.org/conference/osdi18/presentation/zhang.

[77] Tobias Ziegler, Carsten Binnig, and Viktor Leis. Scalestore: A fast and cost-efficient storage engine using dram, nvme, and rdma. SIGMOD '22, page 685–699. Association for Computing Machinery, 2022. https://doi.org/10.1145/3514221.3526187.

STYX: Exploiting SmartNIC Capability to Reduce Datacenter Memory Tax

Houxiang Ji[*], Mark Mansi[†§], Yan Sun[*§], Yifan Yuan[††], Jinghan Huang[*], Reese Kuper[*], Michael M. Swift[†], and Nam Sung Kim[*]

[*]University of Illinois Urbana-Champaign [†]University of Wisconsin-Madison [††]Intel Labs

Abstract

Memory optimization kernel features, such as memory deduplication, are designed to improve the overall efficiency of systems like datacenter servers, and they have proven to be effective. However, when invoked, these kernel features notably disrupt the execution of applications, intensively consuming the server CPU's cycles and polluting its caches. To minimize such disruption, we propose STYX, a framework for offloading the intensive operations of these kernel features to SmartNIC (SNIC). STYX first RDMA-copies the server's memory regions, on which these kernel features intend to operate, to an SNIC's memory region, exploiting SNIC's RDMA capability. Subsequently, leveraging SNIC's (underutilized) compute capability, STYX makes the SNIC CPU perform the intensive operations of these kernel features. Lastly, STYX RDMA-copies their results back to a server's memory region, based on which it performs the remaining operations of the kernel features. To demonstrate the efficacy of STYX, we re-implement two memory optimization kernel features in Linux: (1) memory deduplication (`ksm`) and (2) compressed cache for swap pages (`zswap`), using the STYX framework. We then show that a system with STYX provides a 55–89% decrease in 99^{th}-percentile latency of co-running applications, compared to a system without STYX, while preserving the benefits of these kernel features.

1 Introduction

The modern OS offers various kernel features that can improve the overall utilization and/or performance of systems. Among them, memory optimization kernel features, such as memory deduplication, compressed cache for swap pages, and memory compaction to name a few, are devoted to utilizing the limited DRAM capacity of systems more efficiently. These kernel features are attractive to hyperscalers such as Google, Amazon, Meta, and Microsoft for two key reasons. First, DRAM technology scaling has notably slowed down, resulting in stagnant reduction in cost per GB of DRAM. Second, the DRAM capacity needed for datacenter servers has rapidly grown, not only for applications but also for software packages, profiling, logging, and other supporting functions required for efficient deployment of applications (*i.e.*, datacenter memory tax).

These kernel features have been extensively evaluated and enhanced [6, 19, 22, 27, 31, 40, 51]. They have proven to be effective, but they also incur notable deployment costs. Specifically, they are not frequently invoked, but they often perform memory- and/or CPU-intensive operations. They bring kilobytes to megabytes of usually cold data into the server CPU's caches and then make its cores intensively execute simple but repetitive operations on the data, often after disabling kernel preemption. As a result, they cause significant interference especially with co-running memory-intensive/latency-sensitive applications at the server CPU's cores and caches. This leads to a substantial increase in the high-percentile latency of the applications in datacenters (§3).

In this paper, we propose STYX, using SmarTNIC (SNIC) to efficiently manage the datacenter memorY taX (§4). Specifically, STYX makes use of two common Capabilities of SNIC: **(C1)** the RDMA capability to copy the server's memory regions, on which a kernel feature intends to intensively operate, to SNIC memory (① in Figure 1) and **(C2)** the compute capability to offload the intensive operations of kernel features from the expensive server CPU to the cheap SNIC CPU or accelerators (② in Figure 1). **(C1)** prevents the pollution

[§]Mansi and Sun have contributed equally as second authors.

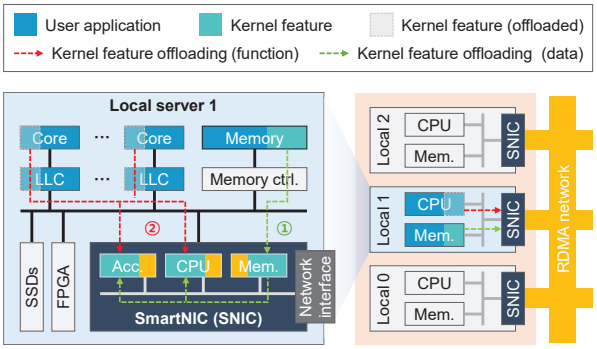


Figure 1: Overview of STYX framework.

of the server CPU's caches that stores application code and data, while **(C2)** frees the server CPU's cores from executing the intensive operations. As such, STYX offers a framework that allows us to deploy kernel features with significantly less disruption to the performance of co-running applications, without making the kernel features less effective.

We choose SNIC as our platform to offload intensive operations of memory optimization kernel features for two **R**easons. **(R1)** SNIC has already been deployed by hyperscalers (*e.g.*, Azure SNIC [15] and Amazon Nitro [4]) to minimize the datacenter tax [24] associated with executing network functions (*e.g.*, compression/decompression, encryption/decryption, and regular expression matching) at high rates. STYX reuses this existing capability without demanding novel or modified hardware. **(R2)** SNIC CPU's cores are not fully utilized as they are typically used to control accelerators in SNIC and orchestrate data transfers between the accelerators and the network controller. Note that STYX is built on a generic RDMA interface. As such, STYX also allows servers with standard RDMA NICs (RNICs) to seamlessly offload the intensive operations of kernel features to other servers with SNICs or RNICs.

To demonstrate the efficacy of STYX, we re-implement two memory optimization Linux kernel **F**eatures as examples: **(F1)** memory deduplication for virtual machines (VMs), also known as kernel same-page merging (`ksm` [14]) and **(F2)** compressed cache for swap pages (`zswap` [21]) (§5). Subsequently, we set up a server with an Intel Xeon CPU-based CPU and an NVIDIA BlueField-2 SNIC [20,37], and take `Redis` [41] driven by `YCSB` [11] as a representative memory-intensive/latency-sensitive application running on datacenter servers (§6). Lastly, we measure the 99^{th}-percentile (p99) response time (or latency) of `Redis` for various cases. (§7).

Specifically, we begin by evaluating the p99 latency values of `Redis` with systems deploying `ksm` (denoted as `sys-ksm`) and `zswap` (`sys-zswap`), and compare them with those of a system that deploys no memory optimization kernel feature (`sys-no-mo`). We show that `sys-ksm` and `sys-zswap` increase the p99 latency values by 4.8–9.7× and 8.1–11.0×, respectively, compared to `sys-no-mo`. Then, we evaluate the p99 latency values of `Redis` with systems deploying STYX-based `ksm` (`sys-styx-ksm`) and `zswap` (`sys-styx-zswap`), and demonstrate that they reduce the p99 latency values to 1.0–1.1× and 1.8–3.8×, respectively, compared to `sys-no-mo`, while preserving the benefits of `ksm` and `zswap`. Finally, we assess the impact of running the STYX-based kernel features on the performance of functions accelerated by SNIC. We choose regular expression matching (`rem`) as a representative function accelerated by a dedicated accelerator in the NVIDIA BlueField-2 SNIC. Even when offloading the intensive operations of `ksm` and `zswap` to the SNIC, STYX increases the p99 latency value of `rem` by only 1.3%.

2 Background

2.1 Memory Optimization Kernel Features

In this section, we provide an overview of two memory optimization kernel features in Linux: `ksm` and `zswap`. Other operating systems, such as Windows, also offer similar features like page combining [7,34] and memory compression [54]).

`ksm`. It is a memory deduplication feature in Linux. It is commonly used with kernel-based virtual machine (KVM) to quickly consolidate more VMs within a given physical memory capacity [35], by sharing pages with the same content among multiple VMs (*e.g.*, pages storing code for OS and common libraries). As it allows for more efficient storage of common data in cache or memory, it also notably improves performance for certain applications and operating systems [47]. It periodically and incrementally scans pages of two or more running processes to identify those with the same memory contents. Then, it merges those identical pages into a single physical copy, updates their page table entries with a copy-on-write (CoW) attribute, and reclaims the memory space previously used by the pages. Both the overhead and benefit of `ksm` are determined by the number of scanned pages per invocation of `ksm`, the frequency of scanning pages, and the maximum number of merged pages.

`zswap`. It serves as a compression backend for the Linux swap daemon (`kswapd`) which includes synchronous direct and asynchronous background paths. `kswapd` takes

the synchronous direct path when the memory allocator fails to allocate pages due to a lack of free memory space. This requires `kswapd` to immediately swap out the least recently used (LRU) pages to the backing swap device. `kswapd` takes the asynchronous background path when the amount of free memory space drops below the `page_low` watermark. This makes `kswapd` begin to swap out pages from the inactive page list, and continues until the amount of free memory space exceeds the `page_high` watermark.

When deployed, `zswap` intercepts the pages from both the paths above, compresses them, and places them in a dynamically allocated memory pool in DRAM (*i.e.*, `zpool`). Meanwhile, when the size of `zpool` reaches the `max_pool_percent` threshold, `zswap` wakes up and takes the LRU page from `zpool`, decompresses and relocates it to the backing swap device, and frees the compressed page from `zpool`. To serve a page fault, `zswap` first checks `zpool` to find whether the page is evicted to the backing swap device. If the page is found in `zpool`, it is simply decompressed and returned by `zswap`. Otherwise, the system follows the standard process for swapping in a page from the backing swap device.

Since `zswap` can notably reduce the need for accessing the slow backing swap device, it may improve the overall performance of the system; the page decompression on the synchronous direct path is part of the performance-critical path for handling page faults, but it is typically faster than retrieving pages from the backing swap device. As such, `zswap` has been evaluated by Google [27] and Meta [51] for potential deployment in production systems.

2.2 SNIC and RDMA

Recently, various SNICs have been developed to offload functions common in network applications such as security, compression, and network function virtualization as part of an effort to reduce the datacenter network processing tax [24]. Generally, an SNIC integrates a traditional network interface controller (NIC) with a CPU, ASIC- and/or FPGA-based accelerators, and memory and IO subsystems. For example, an NVIDIA BlueField-2 SNIC [37] consists of 8 ARM CPU cores with private L1 and shared L2 caches, a cache-coherent on-chip interconnect, DRAM and PCIe controllers, onboard DRAM as main memory, and ASIC-based accelerators for regular expression matching, encryption, and compression. AMD/Xilinx SN1000 [5] integrates an FPGA fabric with an SNIC similar to the NVIDIA BlueField-2 SNIC in a single chip. An SNIC is itself a complete system, recognized as an independent node.

RDMA is supported by most SNICs and standard NICs used in datacenters. As it allows a client to directly access the memory of servers at low latency and high bandwidth, it is now widely used by datacenters [16, 18, 28, 53, 56]. Additionally, an SNIC in a server can access the server's local memory through RDMA. RNIC supports two operating modes: two-sided RDMA and one-sided RDMA. The two-sided RDMA reduces packet processing overhead by delivering requests (or data) from a client directly to server's memory for application processing. One-sided RDMA allows the client to completely bypass the server's CPU and directly read from or write to the server's memory.

3 Impact of Running Kernel Features on Application Performance

A body of prior work has demonstrated that `ksm` and `zswap` have proven to be effective in improving the overall performance of systems [8, 19, 22, 27, 35, 47, 49]. Nonetheless, such benefits come with costs that have often discouraged system administrators from widely deploying them in datacenters. In this section, we analyze the costs associated with deploying `zswap` as an example.

Figure 2 shows a snapshot of (a) consumed CPU cycles, (b) last-level cache (LLC) miss rate, and (c) response latency of `Redis` that are captured when `zswap`-enabled `kswapd` is invoked. See Section 6 for the detailed evaluation setup and methodology. First, `zswap` increases the consumed CPU cycles by 26.4% during the captured time period (Figure 2(a)). Second, it increases the average LLC miss rate from 4.4% to 49.8% (Figure 2(b)). Lastly, it increases the mean, median, 3^{rd} quartile, and p99 latency values of `Redis` serving the same number of requests by 1.5×, 1.2×, 1.8×, and 2.1× respectively (Figure 2(c)).

Although we show the plots only for `zswap` in Figure 2, we observe that `ksm` also exhibits a similar impact on CPU cycle consumption, LLC miss rate and response latency of `Redis`. Subsequently, we discuss the primary sources of such increases in both `zswap` and `ksm` in detail.

`ksm`. It periodically scans pages and calculates a 32-bit checksum for each scanned page to more efficiently identify candidate pages for future merging. Among the candidate pages, `ksm` picks two pages and performs a byte-by-byte comparison to determine if the two pages can be merged. Both the checksum calculation and byte-by-byte comparisons of pages are CPU- and memory-intensive as they bring around 400MB of data to caches and CPU cores (often from DRAM) and do numerous

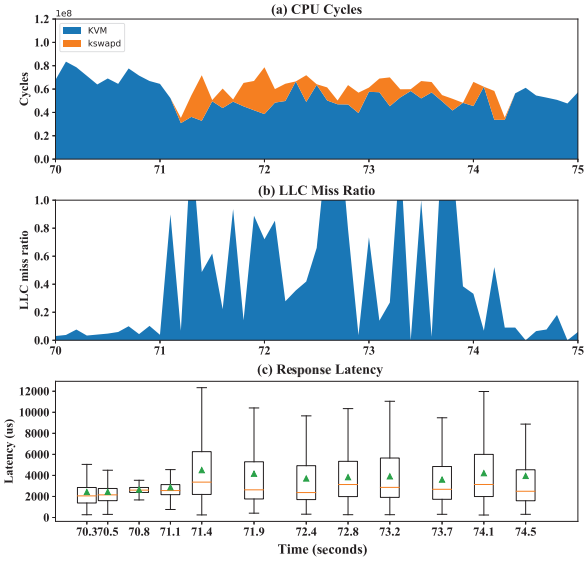


Figure 2: A snapshot of (a) consumed CPU cycles, (b) LLC miss ratio, and (c) response latency before and after invoking `kswapd` while running `Redis`. We gather the response latency values for every 1110 requests and plot them using a box, a vertical line, a triangle, and a horizontal line. The box, triangle, and horizontal line respectively represent the 1^{st} to 3^{rd} quartile range, mean, and median of the latency values for 1110 requests.

arithmetic and comparison operations for that amount of data per invocation of `ksm`.

`zswap`. When invoked, it performs compression and decompression, which are highly compute-intensive and thus significantly consume CPU cycles [23, 39]. For instance, in Figure 2, approximately 45,000 pages are compressed in only 5 seconds, consuming roughly 25–50% cycles of the server CPU's core while `zswap` is running. These pages represent 175MB (*i.e.*, 45,000 × 4KB) of *cold* data that is brought into the server CPU's LLC. Since they are unlikely to be used soon, they end up polluting the server CPU's LLC. Later, when compressed pages in `zpool` are evicted to the backing swap device (§2.1), the pages are decompressed and re-pollutes the server CPU's LLC with *cold* data again.

4 STYX Framework

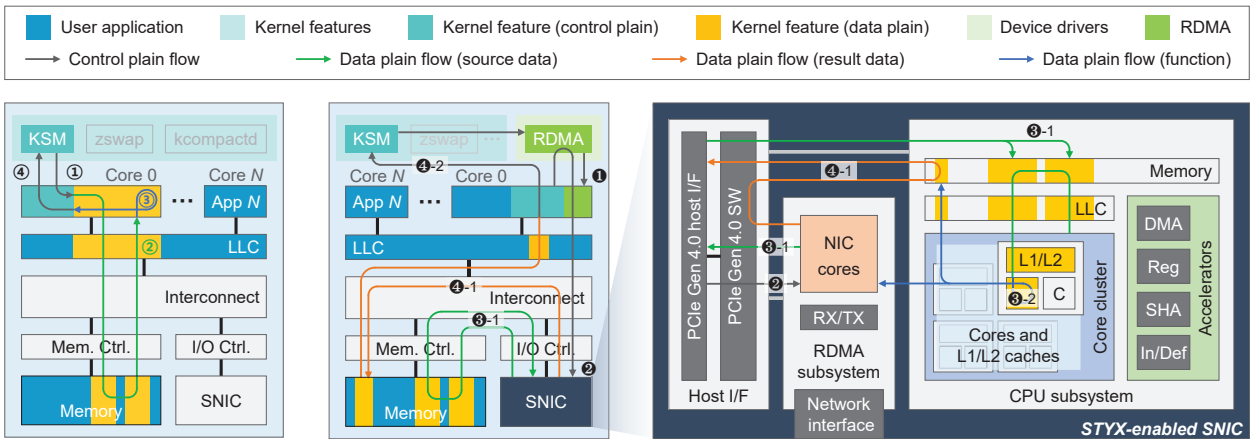
In Section 3, we demonstrated that widely used memory optimization kernel features are often CPU- and memory-intensive, and significantly interfere with co-running applications at the server CPU's cores and caches. To reap the benefits of deploying these kernel features while minimizing interference with the co-running applications, we propose STYX. In this section, we provide an overview of STYX and describe its workflow as a general framework. Subsequently, in Section 5, we delve into the usage of STYX for offloading CPU- and memory-intensive operations of `ksm` and `zswap` to SNIC.

4.1 Overview

We design STYX based on a key observation that memory optimization kernel features, similar to network applications, can be decomposed into control and data planes. We then assign the most CPU- and memory-intensive operations of the kernel features to the data plane, and have the SNIC's CPU handle the data plane. This facilitates STYX to significantly reduce the costs of deploying the kernel features without compromising their benefits.

Figure 3a depicts an abstracted workflow of conventional memory optimization kernel features. When invoked, a kernel feature running on the server's CPU ① determines one or more memory regions that it intends to operate on; ② copies the memory regions to the server CPU's caches; ③ operates on the memory regions (*e.g.*, comparing two pages using `memcmp` in the case of `ksm`); and ④ makes a decision for the next step (*e.g.*, whether merge two pages or not in the case of `ksm`) based on the result of ③. STYX considers ① and ④ as the control plane, while it assigns ② and ③ to the data plane.

In a conventional system, the server's CPU performs both control- and data-plane operations of a given kernel feature. As discussed in Section 3, the data-plane operations significantly pollute the server CPU's caches and intensively consume the server CPU's valuable cycles. In contrast, in an STYX-based system, the SNIC's CPU performs data-plane operations instead, while the server's CPU still performs the control-plane operations. Specifically, the control plane on the server's CPU first determines memory regions that the data plane on the SNIC's CPU will operate on, RDMA-copies the memory regions from the server's memory to the SNIC's memory, and then makes the data plane on the SNIC's CPU operate on the memory regions. After the data plane on the SNIC's CPU completes operations on the RDMA-copied memory regions, it RDMA-copies the results back to the server's designated memory region. Finally, as the conventional system does, the control plane on the server's CPU decides the next step based on the results.



(a) Conventional workflow

(b) Workflow of kernel feature offloading via STYX framework

Figure 3: Workflow of a memory optimization kernel feature in conventional and STYX-enabled systems. For brevity, only the cache and memory regions of the kernel feature's data plane are illustrated.

```
struct STYX_descriptor {
  func_type func;
  // memory regions RDMA-copied to SNIC
  u64* addrs; // starting addresses
  int* lens;  // lengths
  int  num;   // number
  // RDMA resources
  void* completion_queue;
  void* send_queue;
  void* recv_queue;
  ...
};
```

Listing 1: Data structure of `STYX_descriptor`.

4.2 Workflow

Figure 3b illustrates a high-level workflow of STYX framework, which is built on top of kernel-space RDMA verbs. It comprises four steps: ❶ setup, ❷ submission, ❸ remote execution, and ❹ completion.

❶ **Setup.** STYX first determines the functions that will be offloaded to SNIC. A function comprises data-plane operations within a specific kernel feature. For instance, `memcmp`, which performs a byte-by-byte comparison of two memory regions, can be such a function in `ksm`. Next, STYX establishes a communication interface between the server and the SNIC by setting up RDMA connections between them. Specifically, STYX allocates necessary RDMA resources, such as completion and work queues, in the kernel space on the server and the user space on the SNIC (❶ in Figure 3(b)). To avoid contention for the RDMA resources among functions, STYX sets up one RDMA connection for each function. Finally, for each function, STYX creates two descriptors, each called `STYX_descriptor`, on the server and the SNIC, respectively, and then associates the descriptors with the corresponding RDMA connection. `STYX_descriptor` is a data structure described in Listing 1, which stores the following information: a function identifier, pointers to the starting addresses of memory regions, the lengths and number of the memory regions, and pointers to the RDMA resources. It is designed to provide a uniform and generic interface for a server to provide necessary information for an SNIC that will execute a specific function on behalf of the server.

❷ **Submission.** Before a kernel feature executes a function, STYX on the server updates the descriptor associated with the function with the starting addresses and lengths of memory regions that it has determined to work on. Next, STYX uses two-sided RDMA to offload the function. Specifically, STYX on the server sends an RDMA `send` request based on the updated descriptor, making the SNIC RDMA-copy the memory regions from the server's memory to the SNIC's memory (❷ in Figure 3(b)). Lastly, STYX calls RDMA `recv`. This suspends the execution of the kernel feature until the SNIC sends the results of the function back to the server through RDMA `send`, and allows the kernel feature to yield the server CPU's core to other application processes.

Alternatively, STYX can employ one-sided RDMA. In this approach, STYX on the server posts the updated starting addresses and lengths to the server's designated memory region (*i.e.*, the descriptor on the server) registered to the SNIC. At the same time, STYX on the SNIC continuously polls the memory region using RDMA `read`. Once STYX on the SNIC obtains the updated addresses and

lengths, it proceeds with RDMA-copying the memory regions from the server's memory to the SNIC's memory using RDMA `read`. This one-side RDMA-based approach can allow STYX to free up the server's CPU for other application processes faster than the two-sided RMDA-based approach. However, it requires the SNIC's CPU to poll the registered memory region using RDMA `read` requests, resulting in consuming PCIe interconnect bandwidth and SNIC CPU's cycles.

❸ **Remote Execution.** After receiving an RDMA `send` request from the server, the SNIC starts to RDMA-copy the server's memory regions to the SNIC's memory region pointed by the RDMA `recv` request using a single scatter-gather transfer (❸-1 in Figure 3(b)). The completion of serving the RDMA `recv` request wakes up STYX on the SNIC to execute a function associated with the RDMA connection. Subsequently, STYX on the SNIC creates a worker thread to execute the function which operates on the RDMA-copied memory region (❸-2 in Figure 3(b)). Note that executing such a function may interfere with network applications co-running on the SNIC. Nonetheless, the SNIC CPU serves as a control plane for network applications executed by the SNIC accelerators (§1), and STYX utilizes unused or underutilized SNIC CPU cores. Therefore, STYX negligibly affects the performance of network applications running on the SNIC (§7.6).

❹ **Completion.** After completing the remote execution of the function, STYX on the SNIC posts an RDMA `send` request to send the results (*e.g.*, the checksum of a page in the case of `ksm`) to STYX on the server (❹-1 in Figure 3(b)). After STYX on the server receives the result through RDMA `recv` previously invoked at ❷, it makes the kernel feature resume the execution and read the results from the server memory (❹-2 in Figure 3(b)). At the same time, STYX on the SNIC invokes RDMA `recv`, which makes STYX on the SNIC sleep until it receives RDMA `send` from the server.

Similar to what is discussed in ❷, STYX on the SNIC can send the result to the server through one-sided RDMA `write` to a memory region registered on the server. However, this demands the server's CPU to keep polling the memory region until the completion signal is detected. This not only wastes the server CPU's cycles but also prevents the kernel feature from yielding the server CPU to application processes.

5 Offloading Kernel Features with STYX

In Section 4, we provided a high-level workflow of STYX as a general framework for offloading intensive operations (or functions) of memory optimization kernel features to SNIC. In this section, we will further elaborate on implementations of STYX-based `ksm` and `zswap`, as well as optimizations tailored for each kernel feature.

5.1 `ksm`

`ksm` is a memory-deduplication feature that merges pages with the same content. We identify the two most resource-intensive functions to offload to SNIC: (1) page comparison and (2) checksum calculation. The page comparison gives the relative address of the first byte that differs in two pages. This is used to determine whether the pages can be merged and the relative order of the two pages. The checksum calculation provides a word-size hash value calculated based on the page content and indicates whether a page has been changed between scan passes by the `ksm` daemon. Algorithm 1 describes one pass of STYX-based `ksm` where the page comparison and the checksum calculation are performed by `STYX_compare` and `STYX_checksum`, respectively.

Since we offload two functions to SNIC, STYX creates two RDMA connections and two descriptors during the setup phase (❶). `STYX_compare` requires two for the number of memory regions as it compares two pages (or memory regions). On the other hand, since

Algorithm 1: `ksm` with STYX offloading

```
1  Init stable_tree and unstable_tree
2  while pages for this pass > 0 do
3      cand_page = next page in the pass
4      for page ∈ stable_tree do
5          if STYX_compare(cand_page, page) then
6              merge(cand_page, page)
7              goto line 2
8      new_cksum = STYX_checksum(cand_page)
9      old_cksum = cand_page.cksum
10     cand_page.cksum = new_cksum
11     if new_cksum == old_cksum then
12         for page ∈ unstable_tree do
13             if STYX_compare(cand_page, page)
                 then
14                 merged_page = merge(cand_page,
                     page)
15                 cow_protect(merged_page)
16                 remove(page, unstable_tree)
17                 insert(merged_page, stable_tree)
18                 goto line 2
19         insert(cand_page, unstable_tree)
20 End of pass, sleep()
```

STYX_checksum calculates a checksum for a given page, it needs one for the number of memory regions. For the length of a memory region, both STYX_compare and STYX_checksum use the size of a page in byte. During the submission phase (❷), STYX updates the starting addresses of the memory regions in the descriptor for STYX_compare with the pointers to two chosen pages. For STYX_checksum, STYX takes the pointer to a selected page and updates the descriptor accordingly. Subsequently, STYX RDMA-copies these memory regions from the server memory to the SNIC memory. During the remote execution phase (❸), STYX on the SNIC performs a byte-by-byte comparison and an xxHash-based checksum calculation [10] on the RDMA-copied page(s). It then returns the relative address of the first byte that differs in the two pages (STYX_compare) and the word-size checksum (STYX_checksum), respectively, to STYX on the server. Receiving the results from STYX on the SNIC, STYX on the server decides whether it will merge the two pages or not, and updates the checksum value for the scanned page during the completion phase (❹).

5.2 zswap

zswap serves as a compression backend for kswapd, and it was incorporated into the Linux kernel starting from version 3.5. As described in Section 2.1, there are synchronous direct and asynchronous background paths.

Algorithm 2: kswapd with STYX offloading

```
1  while kswapd_enabled do
2      if free_page < page_low then
3          kswapd_running = true;
4          while kswapd_running do
5              page = page_to_swap_out()
6              if zpool > max_zpool_size then
7                  if STYX_decompression(LRU_page,
                     dst) fails then
8                      kernel_decompress(LRU_page,
                         dst);
9                  write_to_backing_swap_device(dst);
10                 free_zpool_space(LRU_page);
11             if STYX_compression(page, dst) fails
                 then
12                 kernel_compress(page, dst);
13             write_to_zpool(dst);
14             if free_page > page_high then
15                 kswapd_running = false;
16     else
17         kswapd_sleep();
```

STYX is capable of offloading functions from both paths to SNIC. Nonetheless, as an optimization, we choose to offload only the asynchronous background path which is taken when (1) the amount of free memory space falls below the page_low watermark, and (2) the size of zpool reaches the max_pool_percent threshold. Specifically, when (1) happens, STYX-based zswap makes SNIC compress pages and place the compressed pages in zpool until the amount of free memory space is above the page_high watermark. When (2) occurs, it makes SNIC decompress the LRU page from zpool and relocate it to the backing swap device. We propose this optimization because the latency involved in RDMA-copying pages to the SNIC memory over the PCIe interconnects (*i.e.*, $\sim 5\mu$s) may slow down the time-sensitive synchronous direct path, and degrade overall system performance. Algorithm 2 describes kswapd modified to support STYX-based zswap where the page compression and decompression are performed by STYX_compression and STYX_decompression, respectively.

Since we offload two functions to SNIC, the setup and submission phases for STYX-based zswap are exactly the same as STYX-based ksm except that zswap has only one memory region to RDMA-copy to the SNIC memory for both the functions. Finally, after the remote execution of STYX_compression and STYX_decompression, STYX on the SNIC will return the compressed and decompressed pages, respectively, to STYX on the server.

6 Methodology and Implementation

System Setup. We set up a server with an Intel Xeon Gold CPU and an NVIDIA BlueField-2 SNIC. The detailed hardware and software configurations of the server are listed in Table 1. Note that we lock the CPU frequency at 2.1 GHz and disable hyper-threading (HT) for more consistent performance over multiple measurement runs. VMs are pinned to specific CPU cores to reduce performance variations and interference caused by dynamic voltage/frequency scaling (DVFS) [12], HT [32], and VM scheduling.

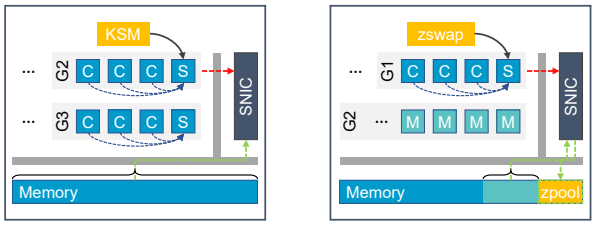
Workload. We run Redis [41] with Yahoo! Cloud Serving Benchmark (YCSB) [11] on the system. Redis is an in-memory data store and is used as a distributed, in-memory key–value database, cache, and message broker, with an optional durability feature. YCSB is a benchmarking framework to evaluate the performance of various in-memory key-value stores. It comes with four workloads: (a) update heavy, (b) read heavy, (c) read only, and (d) read latest that consist of (a) 50% read and 50% update, (b) 95% read and 5% update, (c) 100% read, and

Table 1: Hardware and Software configurations.

Intel Xeon 6138P Server
CPU: 16 Skylake cores @ 2.1GHz w/ HT disabled, 32KB L1, 1MB L2, and 1MB L3 caches per core **Memory:** 5-Ch. w/ 5 16GB DDR4-2666 DRAM modules **OS:** Ubuntu 18.04.6 LTS, Linux kernel 5.4
NVIDIA BlueFeild-2 SNIC
Network: ConnectX-6 Dx w/ two 25 Gbps Ethernet ports , RDMA over converged Ethernet V2 **CPU:** 8 ARM A72 cores @ 2.5GHz, 640 KB L1 per core, 4 MB L2 caches per 2 cores, and 6 MB L3 cache **Memory:** 1 Ch. w/ 16GB DDR4-1600 DRAM module **Accelerators**: regular expression matching, compression, and cryptography **OS**: Ubuntu 20.04.2 LTS, Linux kernel 5.4
Kernel Feature
ksm: `sleep_between_scan`=20ms, `free_mem_thres`=20 `pages_to_scan` ∈ [64, 1250] # adjusted by *ksmtuned* **zswap**: `compressor_type` = lzo, `max_pool_percent` = 20 `zpool_management` = zbud
Virtual Machine
Hypervisor: QEMU-KVM 2.11.1 **VM**: Ubuntu Cloud 18.0, 1 Core, 4GB memory

(d) 95% read and 5% insert, respectively.

Methodology. Figure 4 depicts the evaluation environments for `ksm` and `zswap`. `ksm` aims to reduce memory usage in virtualized environments where multiple VMs are running similar workloads. We set up 16 VMs and pin each VM to a physical core. Then, we organize the VMs into 4 groups, each comprising 3 VMs for `Redis` clients and 1 VM for a `Redis` server. To trigger `zswap`, we set up a background workload designed to allocate



(a) Setup for **`ksm`** (b) Setup for **`zswap`**

Figure 4: Experimental setup for evaluating STYX-based `ksm` and `zswap`. The blue-color boxes indicate the Redis server ('S') and client ('C'). The lighter-blue blocks ('M') represent the cores running a background workload.

and free memory space periodically. We need such a background workload because `Redis` is in-memory data store, and it should be configured not to incur any page faults. Otherwise, p99 latency is dominated by handling page faults. We use `cgroup` [1] to protect the pages used by `Redis` from being swapped out. In the experimental setup for `zswap`, `Redis` servers and clients run on the physical cores directly without VMs.

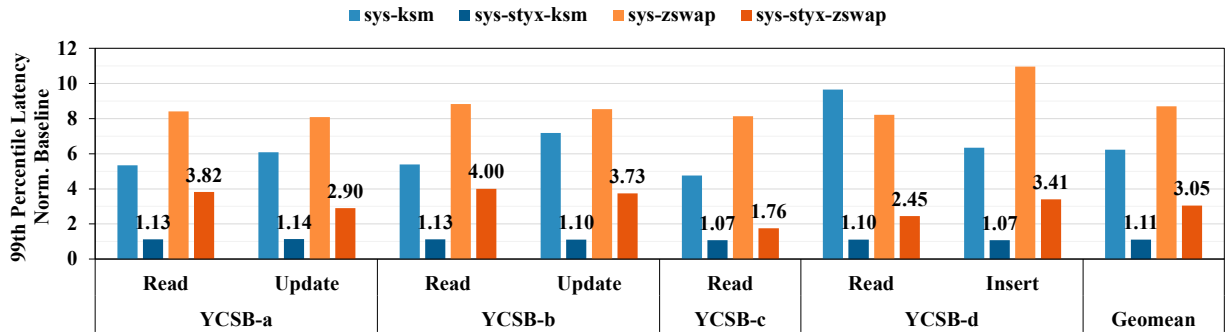
7 Evaluation

7.1 Latency

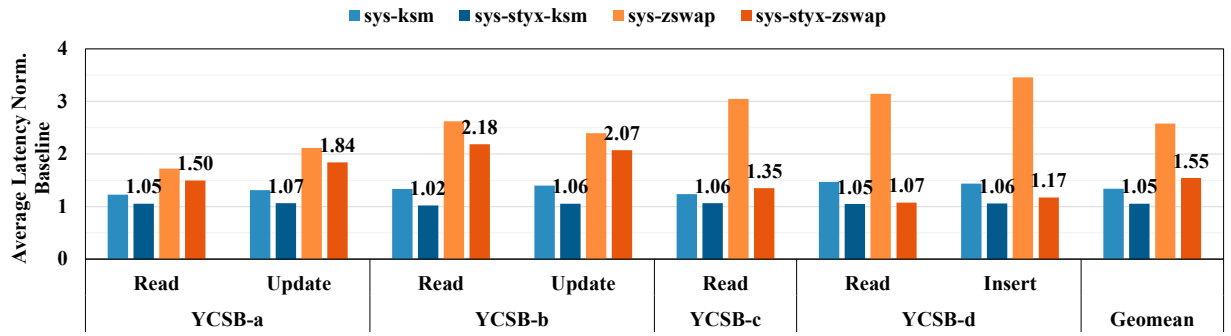
In this section, we choose p99 latency as a key performance metric for our evaluation because many important datacenter applications need to meet certain high-percentile latency requirements [36, 38]. Figure 5a shows the p99 latency values of `Redis` on systems that deploy `ksm` (`sys-ksm`), `zswap` (`sys-zswap`), STYX-based `ksm` (`sys-styx-ksm`), and STYX-based `zswap` (`sys-styx-zswap`), normalized to those of a system without deploying any memory optimization kernel feature (`sys-no-mo`). Overall, it demonstrates that STYX can significantly reduce the p99 latency increased by deploying `ksm` and `zswap`.

Specifically, on average (geometric mean), `sys-ksm` and `sys-zswap` give 6.24× and 8.70× higher p99 latency values than `sys-no-mo`, respectively. In contrast, `sys-styx-ksm` and `sys-styx-zswap` offer 1.11× and 3.05× higher p99 latency values than `sys-no-mo`, respectively. That is, `sys-styx-ksm` and `sys-styx-zswap` reduce the p99 latency increase by 5.62× and 2.85×, compared to `sys-ksm` and `sys-zswap`, respectively. Note that `sys-styx-ksm` offloads most of the intensive operations to SNIC, practically eliminating the p99 latency increase of `sys-styx-ksm`. On the other hand, `sys-styx-zswap` offloads only the intensive operations of the asynchronous background path to SNIC. That is, the intensive operations of the synchronous direct path still affect the p99 latency of `Redis`, contributing to more than 3× increase in p99 latency.

In addition, Figure 5b shows the average latency values of `Redis` on those systems. The average latency of `Redis` is also an important performance metric, as it is inversely proportional to the throughput. Note that `Redis` often throttles serving requests to prevent response latency from increasing too much when the system receives more requests than it can handle efficiently [3]. On average, `sys-ksm` and `sys-zswap` give 1.34× and 2.58× higher average latency than `sys-no-mo`, respectively. In contrast, `sys-styx-ksm` and `sys-styx-zswap`



(a) p99 latency



(b) Average latency

Figure 5: Latency values of `Redis` with `YCSB` workloads running on `sys-ksm`, `sys-styx-ksm`, `sys-zswap` and `sys-styx-zswap`, normalized to `sys-no-mo`.

offers only 1.05× and 1.55×, higher average latency than `sys-no-mo`, respectively. That is, `sys-styx-ksm` and `sys-styx-zswap` reduce the average latency increase by 22% and 40%, compared to `sys-ksm` and `sys-zswap`, respectively.

7.2 LLC Miss Rates

To analyze how STYX reduces the negative impact of deploying `ksm` and `zswap` on p99 latency of `Redis`, we measure the LLC miss rates of the server CPU every 1 second while evaluating `sys-no-mo`, `sys-styx-*` and `sys-*`. We report the p99 LLC miss rates from approximately 160 1s intervals instead of the average LLC miss rates, because intervals with high LLC miss rates are likely responsible for p99 latency values of `Redis`.

Table 2 summarizes the p99 LLC miss rates across all `YCSB` workloads at their highest throughput values that the systems can provide. This shows that the memory optimization kernel features can significantly increase the p99 LLC miss rates, bringing large amounts of cold data into the server CPU's caches when invoked. Specifically, `sys-ksm` and `sys-zswap` give 7.33× and 1.70× higher p99 LLC miss rates than `sys-no-mo`, respectively, in some intervals. In contrast, `sys-styx-ksm` and `sys-styx-zswap` offer 3.78× and 1.28× higher p99 LLC miss rates than `sys-no-mo`, respectively. That is, `sys-styx-ksm` and `sys-styx-zswap` reduce the p99 LLC miss rate increase by 48% and 25%, respectively. Such a benefit comes from the fact that `sys-styx-ksm` and `sys-styx-zswap` RDMA-copy the server's memory regions that `ksm` and `zswap` work on to the SNIC memory instead of the server CPU's caches.

Table 2: p99 LLC miss rates of three systems (`sys-no-mo`, `sys-*`, and `sys-styx-*`) for different `YCSB` workloads.

	a	b	c	d	GeoMean
`no-mo`	9.7%	7.1%	7.3%	8.0%	8.0%
`ksm`	60.4%	56.9%	59.8%	57.5%	58.6%
`styx-ksm`	40.4%	26.5%	27.2%	28.4%	30.2%
`no-mo`	18.5%	21.4%	22.2%	21.7%	20.9%
`zswap`	34.7%	41.3%	33.9%	32.6%	35.5%
`styx-zswap`	25.1%	27.8%	29.8%	24.7%	26.8%

Table 3: CPU utilization of two systems (sys-* and sys-styx-*) for different YCSB workloads.

	a	b	c	d	GeoMean
ksm	26.0%	26.0%	25.9%	25.9%	26.0%
styx-ksm	7.1%	7.3%	6.8%	6.7%	7.0%
zswap	23.5%	19.8%	20.5%	17.8%	20.3%
styx-zswap	13.0%	8.9%	11.8%	8.4%	10.4%

Note that the p99 LLC miss rate of sys-no-mo for ksm is much lower than that of zswap. This is because of the background workload designed to incur page faults for zswap. Besides, in the case of zswap, STYX offloads only intensive operations of the asynchronous background path to SNIC. That is, intensive operations of the synchronous direct path still pollute the server CPU caches when the background workload incurs page faults. Lastly, even in the case of ksm, STYX does not offload all the operations, either.

7.3 CPU Cycle Consumption

In addition to reducing cache pollution, STYX conserves the server CPU's cycles, as it offloads the CPU-intensive operations to the SNIC CPU. To assess the impact of running ksm and zswap on consuming the server CPU's cycles, we identify the number of 1-millisecond intervals that both a kernel feature and Redis co-run on a server CPU's core while measuring the number of server CPU's core cycles consumed by the kernel feature during these intervals. To get the average CPU utilization shown in Table 3, we sum up all the server CPU's core cycles consumed by the kernel feature and then divide it by the total number of the server CPU's core cycles during the intervals in which the kernel feature and Redis co-run.

Table 3 shows that STYX considerably reduces the consumption of the server CPU's core cycles. On average, sys-styx-ksm and sys-styx-zswap reduce the server CPU's core cycles consumed by ksm and zswap from 26% to 7% and from 20% to 10%, respectively. The server CPU's cycles saved by offloading intensive operations of ksm and zswap to SNIC can be used for Redis, which minimizes disruption of Redis operations during these co-running intervals.

7.4 Offloading Latency

Table 4 shows the breakdown of the latency values of ksm and zswap functions offloaded to the NVIDIA BlueField-2 SNIC. By analyzing the breakdown, we can identify the offloading steps that may provide further optimization opportunities. We do not include the latency breakdown of the setup step (❶), because it is called once and the latency cost is amortized over time. The submission step (❷) takes ∼0.5 microseconds, *e.g.*, only ∼2% of the total latency of offloading the functions to SNIC. This latency primarily comes from the time to send an RDMA send request to SNIC.

Table 4: The breakdown of the offloading latency values of each function and the percentage values of function execution time in total kernel feature execution time per invocation. f1 and f2 correspond to comparison and checksum of ksm, respectively. For f1, we measure the latency of comparing two pages with the same content, which gives the longest latency. f3 and f4 represent compression and decompression of zswap, respectively.

		f1	f2	f3	f4
styx-ksm/zswap	❷ (μs)	0.51	0.49	0.52	0.49
	❸ (μs)	14.61	12.93	20.26	16.97
	❹ (μs)	5.04	4.97	5.21	5.13
	% in Tot.	57.2	32.3	25.4	8.3
ksm/zswap	% in Tot.	36.9	19.5	12.3	6.1

The remote execution step (❸) takes a total of 13–20 microseconds depending on the offloaded functions. Specifically, it spends 5–7 microseconds for RDMA-copying memory regions from the server memory to the SNIC memory. It spends the remaining 8–15 microseconds for the SNIC CPU to execute the functions of ksm and zswap. As the RDMA-copy latency is responsible for a dominant fraction of the remote execution step, we may consider making SNIC's on-chip accelerators execute these functions to further reduce the remote execution latency. However, the accelerators in the NVIDIA BlueField-2 SNIC are connected to the on-chip PCIe interconnect. Thus, it still takes a notable amount of time to offload functions from the SNIC CPU to the accelerators (*e.g.*, ∼7 milliseconds for the compression accelerator), which involves another DMA transfer within SNIC.

The completion step (❹) consumes a notable amount of time spent by interrupt handling and process context switching between application and kernel feature processes. During this step, the SNIC CPU remains active after submitting the RDMA request until receiving an acknowledgment from the server CPU. This waiting time is included as part of the latency of the completion step.

7.5 Effectiveness of Kernel Features

It takes a longer time to offload functions of the kernel features to SNIC than to run them directly on the server CPU. This in turn increases the overall execution time

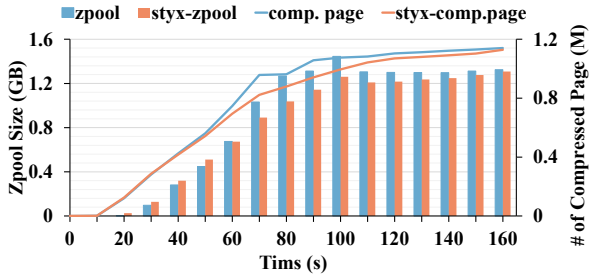


Figure 6: The number of the compressed pages (line) and the size of `zpool` (area) over time in `sys-zswap` and `sys-styx-zswap`.

of the kernel features and may affect the kernel feature's effectiveness. We take `zswap` as an example and plot: (1) the number of compressed pages and (2) the size of `zpool` over time in Figures 6. Although the compression operation performed by the SNIC in STYX does result in longer latency, the overall effectiveness and performance of `zswap` are not greatly affected. First, we observe that the size of `zpool` is not directly proportional to the number of compressed pages. This is because the compression ratio of pages varies across pages. Figure 6 shows that STYX-based `zswap` can provide a comparable compression rate close to what `zswap` can, based on the number of compressed pages. Figure 6 also shows that with STYX, the rate of growth of `zpool` is only 2% lower than that of the standard `zswap` implementation. This is explained by run-to-run performance variations.

Note that both `zswap` saves the disk I/O as it compresses the pages into swap memory and avoids the direct swap out. We monitor the disk I/O at the runtime of the workloads and see the overall disk I/O consumption is 39% lower when `zswap` is enabled. The disk I/O reduction is attributed to the swap-out compressing cache in `zswap`. Upon deploying STYX, the disk I/O throughput remains 35% lower than the `sys-no-mo` case where no `zswap` feature runs. We see that both `sys-zswap` and `sys-STYX-zswap` achieve comparable disk I/O saving. In summary, STYX-based `zswap` preserves the benefits of `zswap` while notably reducing the disruption of co-running applications caused by `zswap`.

7.6 SNIC Application Performance

Since SNIC has its own designated roles, we need to analyze the impact of running STYX on performance of applications that SNIC is designed to accelerate. Specifically, we choose regular expression matching (`rem`) as an application; since it has been extensively used for various network security applications and the NVIDIA BlueField-2 SNIC provides a dedicated accelerator. Table 1 gives an overview of the NVIDIA BlueField-2 SNIC. We take the `DPDK-Pktgen` tool [52] on a remote server to send network packets to the SNIC. We configure the SNIC and `DPDK-Pktgen` to exercise the maximum 25Gbps network bandwidth, and vary the size of packets to observe the utilization of the SNIC CPU's cores.

As the packet size increased from 128 bytes to 1024 bytes, the number of SNIC CPU's cores required to achieve the maximum `rem` throughput decreases from 5 to 1. Smaller packet sizes demand more packets per second to use the full network bandwidth, which in turn requires more cores. In our current implementation, STYX on the SNIC utilizes only ∼30% of a single core of the SNIC CPU, which is obtained after running the most CPU-intensive function, page compression in `zswap`. That is, the SNIC can handle STYX with little impact on the performance of `rem`. Our experiment shows that the SNIC running only `rem` gives a p99 latency of 13.83 microseconds, while the SNIC running both `rem` and STYX offers a p99 latency of 13.85 microseconds. It also confirms that the SNIC running both `rem` and STYX do not decrease the maximum throughput of `rem`.

8 Related Work

Exploring the improved compute efficiency of heterogeneous computing, many past proposals have focused on offloading CPU-intensive operations of user-space programs from the CPU to xPUs and FPGA. In contrast, relatively less attention has been given to offloading CPU- and memory-intensive operations of kernel-space programs from the CPU so far. Nevertheless, it has become increasingly important, especially for datacenter servers to cost-effectively reduce the high datacenter tax.

Some past proposals aim to make kernel features run more efficiently. One pioneering proposal is `Pageforge` [45], which implements a hardware mechanism in the memory controller to execute the page comparison operations of `ksm`. It also exploits the Error Correcting Code (ECC) engine in the memory controller to perform the checksum calculation operations of `ksm`. Although `Pageforge` is effective, it requires hardware changes in the memory controller. Lin *et al.* propose to accelerate checksum calculation using GPU [30]. `XLH` [35] enhances the page scan in `ksm`, utilizing hints from guest I/O in a VM environment. It allows `ksm` to identify mergeable pages earlier and merge more pages. Nonetheless, it does not reduce either the consumption

of the server CPU's cycles or the pollution of its caches. `ezswap` [25] estimates the compression ratios of pages in advance, compresses only highly-compressible pages, and store them in `zpool`. Classic `zswap` blindly chooses pages to swap out and compress based on the LRU policy. Song *et al.* propose an efficient way to enhance `zswap` by skipping the compression of incompressible pages [46]. These optimizations are orthogonal to our work and can be employed together with STYX.

Abeyrathne *et al.* [2] demonstrated the potential of offloading kernel functions to FPGA by utilizing the advanced features of the latest Xilinx FPGA with the provided kernel modules. STYX instead deals with the problems by elaborative and creative designs without any limitation on the FPGA model. Roulin *et al.* [42] examined the migration of the user-space network switch daemon to the kernel space. This setup aims to grant complete control of the routing ASIC to the Linux kernel, thereby reducing the overhead of kernel-space and user-space communication. However, this approach does not offer a general solution to the communication between the OS kernel and the offloading device, as it is specifically designed for network switch APIs.

There also have been many studies conducted on SNIC to explore its capacity in various ways. Offloading various functions, such as distributed services and intrusion detection, to SNICs is a promising approach to mitigate resource consumption on servers, enhance the performance of specialized operations, and improve overall energy efficiency [9, 13, 15, 17, 29, 48, 50]. Specifically, `LineFS` [26] offloads distributed file system. `FlexTOE` [44] offloads TCP to SmartNIC with flexibility and high performance. `Xenic` [43] uses the LiquidIO 3 SNIC [33] for fast distributed transactions. `Pigasus` [55] uses an FPGA-based SNIC to accelerate intrusion detection and prevention systems. STYX focuses on harnessing the capabilities of SNICs to effectively mitigate the datacenter memory tax.

9 Conclusion

In this paper, we first showed that memory optimization kernel features intensively consume the server CPU's cycles and pollute its caches when they are invoked. This in turn leads to a significant increase in the p99 latency of memory-intensive/latency-sensitive datacenter applications. Second, we proposed STYX as a solution to minimize the consumption of server CPU's cycles and the pollution of its cache caused by these kernel features. STYX accomplished these by leveraging the RDMA and compute capabilities of modern SNIC, and offloading the intensive operations of these kernel features to SNIC. Lastly, we demonstrated the effectiveness of STYX after re-implementing two memory optimization kernel features in Linux: `ksm` and `zswap` using the STYX framework and running memory-intensive/latency-sensitive applications. We showed that the systems with STYX-based `ksm` and `zswap` achieved 5.6× and 2.9× lower p99 latency values than the systems with classic `ksm` and `zswap`, while preserving the benefits of `ksm` and `zswap`.

Acknowledgments

We thank Jiacheng Ma and Ipoom Jeong for their technical discussion and support. This work was supported in part by Samsung Electronics, the IBM-Illinois Discovery Accelerator Institute and PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Nam Sung Kim has a financial interest in Samsung Electronics and NeuroRealityVision.

References

[1] Control Groups (Cgroups) - Linux kernel documentation. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html. 2022.

[2] Pabudi T Abeyrathne, S Devapriya Dewasurendra, and Dhammika Elkaduwa. Offloading specific performance-related kernel functions into an FPGA. In *2021 IEEE 30th International Symposium on Industrial Electronics (ISIE'21)*, 2021.

[3] Alibaba Cloud Database Team. Improving redis performance through multi-thread processing. /https://www.alibabacloud.com/blog/improving-redis-performance-through-multi-thread-processing_594150/, 2018.

[4] Amazon Web Services. Aws re:invent 2018: Powering next-gen ec2 instances: Deep dive into the nitro system (cmp303-r1). https://www.youtube.com/watch?v=e8DVmwj3OEs, 2018.

[5] AMD/Xilinx. Alveo SN1000 smartnic accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/sn1000.html, 2023.

[6] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the linux symposium*, 2009.

[7] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP'16)*, 2016.

[8] Wenqi Cao and Ling Liu. Dynamic and transparent memory sharing for accelerating big data analytics workloads in virtualized cloud. In *2018 IEEE International Conference on Big Data (Big Data'18)*, 2018.

[9] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, 2016.

[10] Yann Collet. xxHash: Extremely fast hash algorithm. https://github.com/Cyan4973/xxHash, 2016.

[11] 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'10)*, 2010.

[12] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC'11)*, 2011.

[13] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[14] Izik Eidus and Hugh Dickins. Kernel Samepage Merging. https://docs.kernel.org/next/admin-guide/mm/ksm.html, 2009.

[15] Daniel Firestone, Andrew Putnam, Hari Angepat, Derek Chiou, Adrian Caulfield, Eric Chung, Matt Humphrey, Kalin Ovtcharov, Jitu Padhye, Doug Burger, Dave Maltz, Albert Greenberg, Sambhrama Mundkur, Alireza Dabagh, Mike Andrewartha, Vivek Bhanu, Harish Kumar Chandrappa, Somesh Chaturmohta, Jack Lavier, Norman Lam, Fengfen Liu, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Kushagra Vaid, and David A. Maltz. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.

[16] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, 2021.

[17] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. 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'20)*, 2020.

[18] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, 2016.

[19] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. SmartMD: A high performance deduplication engine with mixed pages. In *2017 USENIX Annual Technical Conference (USENIX ATC'17)*, 2017.

[20] Jinghan Huang, Jiaqi Lou, Yan Sun, Tianchen Wang, Eun Kyung Lee, and Nam Sung Kim. Analyzing energy efficiency of a server with a smartnic under slo constraints. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'23)*, 2023.

[21] Jennings, Seth. The zswap compressed swap cache. https://lwn.net/Articles/537422/, 2013.

[22] Gangyong Jia, Guangjie Han, Joel JPC Rodrigues, Jaime Lloret, and Wei Li. Coordinate memory deduplication and partition for improving performance in cloud computing. *IEEE Transactions on Cloud Computing*, 7(2):357–368, 2015.

[23] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009.

[24] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd IEEE/ACM International Symposium on Computer Architecture (ISCA'15)*, 2015.

[25] Jongseok Kim, Cheolgi Kim, and Euiseong Seo. *ezswap*: Enhanced compressed swap scheme for mobile devices. *IEEE Access*, 7:139678–139691, 2019.

[26] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM (SOSP'21)*, 2021.

[27] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[28] Yanfang Le, Mojtaba Malekpourshahraki, Brent Stephens, Aditya Akella, and Michael M. Swift. On the impact of cluster configuration on RoCE application design. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking (APNet'19)*, 2019.

[29] Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu. AlNiCo: SmartNIC-accelerated contention-aware request scheduling for transaction processing. In *2022 USENIX Annual Technical Conference (USENIX ATC'22)*, 2022.

[30] Wei-Cheng Lin, Chia-Heng Tu, Chih-Wei Yeh, and Shih-Hao Hung. GPU acceleration for kernel samepage merging. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'17)*, 2017.

[31] Karmen MacKendrick. Fragmentation and memory. In *Fragmentation and Memory*. Fordham University Press, 2022.

[32] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.

[33] Marvell. Marvell LiquidIO III inline dpu based smartnic. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf, 2022.

[34] Microsoft. Cache and Memory Manager Improvements. https://learn.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/cache-memory-management/improvements-in-windows-server, 2022.

[35] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, 2013.

[36] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.

[37] NVIDIA Corporation. NVIDIA BlueField-2 DPU: Data center infrastructure on a chip. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2022.

[38] 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)*, 2019.

[39] Laura Promberger, Rainer Schwemmer, and Holger Fröning. Characterization of data compression across cpu platforms and accelerators. *Concurrency and Computation: Practice and Experience*, page e6465, 2022.

[40] Shashank Rachamalla, Debadatta Mishra, and Purushottam Kulkarni. Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems. In *20th Annual International Conference on High Performance Computing (HiPC'13)*, 2013.

[41] Redislabs. Redis. https://redis.io, 2021.

[42] Andy Roulin. Advancing the state of network switch asic offloading in the linux kernel. Technical report, 2018.

[43] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM (SOSP'21)*, 2021.

[44] Rajath Shashidhara, Timothy Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*, 2022.

[45] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. Pageforge: A near-memory content-aware page-merging architecture. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*, 2017.

[46] Taejoon Song, Myeongseon Kim, Gunho Lee, and Youngjin Kim. Prediction-guided performance improvement on compressed memory swap. In *2022 IEEE International Conference on Consumer Electronics (ICCE'22)*, 2022.

[47] J.P. Stevenson, M.A. Horowitz, D.R. Cheriton, P.M. Hanrahan, and Stanford University. Department of Electrical Engineering. *Fine-grain In-memory Deduplication for Large-scale Workloads*. 2013.

[48] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[49] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 2002.

[50] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. FpgaNIC: An FPGA-based versatile 100gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC'22)*, 2022.

[51] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[52] Keith Wiles. Pktgen-DPDK. https://pktgen-dpdk.readthedocs.io/en/latest/contents.html, 2021.

[53] Jiachen Xue, Muhammad Usama Chaudhry, Balajee Vamanan, T. N. Vijaykumar, and Mithuna Thottethodi. Dart: Divide and specialize for fast response to congestion in RDMA-based datacenter networks. *IEEE/ACM Transactions on Networking*, 28(1), 2020.

[54] Pavel Yosifovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. 2017.

[55] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[56] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)*, 2015.

Change Management in Physical Network Lifecycle Automation

Mohammad Al-Fares[*], Virginia Beauregard[*], Kevin Grant[*], Angus Griffith[*], Jahangir Hasan[*], Chen Huang[*], Quan Leng[*], Jiayao Li[*], Alexander Lin[*], Zhuotao Liu[‡], Ahmed Mansy[*], Bill Martinusen[†], Nikil Mehta[*], Jeffrey C. Mogul[*], Andrew Narver[*], Anshul Nigham[*], Melanie Obenberger[†], Sean Smith[§], Kurt Steinkraus[*], Sheng Sun[*], Edward Thiele[*], and Amin Vahdat[*]

[*]Google
[†]Formerly at Google
[‡]Tsinghua University
[§]Databricks

Abstract

Automated management of a physical network's lifecycle is critical for large networks. At Google, we manage network design, construction, evolution, and management via multiple automated systems. In our experience, one of the primary challenges is to reliably and efficiently manage change in this domain – additions of new hardware and connectivity, planning and sequencing of topology mutations, introduction of new architectures, new software systems and fixes to old ones, etc.

We especially have learned the importance of supporting multiple kinds of change in parallel without conflicts or mistakes (which cause outages) while also maintaining parallelism between different teams and between different processes. We now know that this requires automated support.

This paper describes some of our network lifecycle goals, the automation we have developed to meet those goals, and the change-management challenges we encountered. We then discuss in detail our approaches to several specific kinds of change management: (1) managing conflicts between multiple operations on the same network; (2) managing conflicts between operations spanning the boundaries between networks; (3) managing representational changes in the models that drive our automated systems. These approaches combine both novel software systems and software-engineering practices.

1 Introduction

In large production networks, changes happen all the time. Lots of research and development has delivered a wide range of designs and products for managing changes to network data planes and control planes. Requirements for scalability, reliability, security, low cost, and rapid flexibility together have made it essential to automate many aspects of network management, but this work has mostly focused on managing networks *after* the physical components have been deployed. For example, Software Defined Networking (SDN) methods do not directly address designing the physical wiring of a network, or ensuring that the right switches and cables are ordered from vendors, or connected to effect a design, or how to sequence and schedule this physical work.

At Google, we have also found it necessary to automate many *abstract and physical aspects of a network's full lifecycle*, including network planning (what networks do we need to build and when, given capacity forecasts?), network design (what specific switches and links do we need?), materials ordering (what specific part numbers do we need to order and when, what cables need to be constructed?), network construction (where do data center operators need to place equipment and cables?), firmware installation, physical validation (are the links correctly connected and not suffering high error rates?), network repair processes (which links/switches can we safely drain before doing repairs?), etc. We must also provide our automated control planes with accurate, detailed "schematics" for the networks that they manage.

While initially we could perform *Network Lifecycle Management* (NLM) manually, in practice this was slow, error-prone, and inflexible. Those problems worsened with increasing scale, driving us to automate as much of this work as possible. For example, designing optimal inter-block cabling for a Jupiter fat-tree network is NP-complete, and a good approximation requires significant computation [31, 38]. Even at much smaller scales, processes like correctly rolling out router configuration changes are safest when they are carefully automated [23].

As we introduced systems to automate NLM, we discovered that we had not sufficiently understood or appreciated the difficulty of *change management* in this specific domain. Certainly, change management has always been a problem for network designers and operators, and much useful work has been done on ways to manage changes to device configurations (or SDN controller configurations) related to routing, access control lists, and other post-deployment issues.

However, several other aspects of change management began to delay our progress and undermine the value of our automation. These include managing conflicts between multiple operations on the same network; managing conflicts

between operations spanning the boundaries between networks; managing representational changes in the models that drive our automated systems; and introducing major changes in our software infrastructure.

In this paper, we focus on these change-management challenges, the solutions we developed for them, and some of the experience we gained. In particular, we address several distinct (but interacting) aspects of change management:

Managing conflicts between operations: deciding what order to do things in, and what process steps can be done in parallel without conflicts. The physical lifecycle of a network involves many steps with complex dependencies. In small networks, changes are *sometimes* sufficiently rare and rapid that they can be serialized without loss of efficiency. In a large, frequently-changing network, multiple changes, sometimes with extended execution time, *must* overlap, or else capacity delivery and upgrades becomes unacceptably sluggish (e.g., see §10.1). We must prevent operational conflicts that lead to outages, or even the risk of outages, because we want to do these operations on "live" networks.

Planners must also be able to analyze potential sequences of future changes to decide the best partial order, choose the least costly option, or detect if a sequence would lead to an infeasible or invalid state (we call this "what-if analysis"). Planners also sometimes need to *modify* the order of existing plans, as constraints or requirements change.

Therefore, a key contribution of this paper is the design of a plan-management service (§5.2), and the abstractions that allow us to explicitly represent how various future lifecycle states depend on each other or can be done in parallel.

Representational change: Automation depends on machine-readable data. Foundational to the work in this paper is the MALT network-model representation [27], which we use to represent the current, desired, and potential future states of network topologies at many levels of abstraction. Planning and design processes form a pipeline of successive refinements of these models, and the generation of derived data, such as instructions for creation and placement of cables, from these models. Likewise, operational data, such as device and SDN controller configurations, are primarily derived from these models.

A notable challenge in modeling is that our continued innovation in network designs and components requires us to rapidly evolve the MALT representation (e.g., Fig. 8). Previously, we found it hard to do this safely (without production outages) and without constantly and tediously updating lots of model-generating and model-consuming software; this seriously slowed our innovation.

We describe how we allow a wide variety of model-consuming systems, built and maintained by many different partner teams, to cope with evolution in our MALT representation and how we use it to encode specific designs, without requiring unsustainable engineering efforts on the part of those teams. (§7, §8)

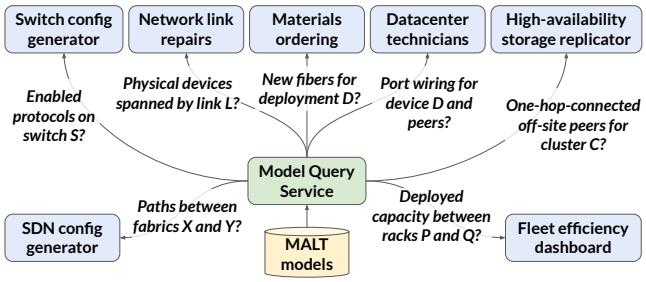


Figure 1: Some use cases for MALT models.

While this paper reports on our experience with large-scale datacenter network infrastructures, we are also applying the same tools and practices in several adjacent domains, such as the management of WAN systems, of machines, and of datacenter physical designs. Our approaches are useful at smaller scales, too.

Ethical concerns: The systems described in this paper do not handle or store any Personal Identifiable Information (PII), and do not raise any ethical issues.

2 Context

This paper focuses on our *Jupiter* datacenter network fabrics [30, 31], and our *B4* WAN fabrics [16, 18]. By "fabric" in this paper, we generally mean a Clos network consisting of many switches and links, with SDN controllers.[1]

We drive our network automation using machine-readable representations, primarily the MALT representation (*Multi-Abstraction Layer Topology*) representation, described in [27], and summarized below in §2.1. We organize these representations as *models* of the network fabric's topology and many other static details.

Our automation covers many aspects of a network's entire lifecycle, including (as depicted in Fig. 1 and Fig. 2):

- **Fabric planning**: Our infrastructure planning team decides when we need to build, expand, or decommission a fabric. They express their high-level goals (where should the fabric be built? Using what abstract architecture? Of what size?) via MALT *fabric intent* models. These models have relatively few details. Planners might explore several possible options for a fabric before committing to a specific choice.
- **Fabric design**: Before we can construct a network, we convert the fabric intent into *concrete* MALT models, via a fully-automated process somewhat like compilation of a program. The concrete models, after several levels of refinement, fully define the structure of the network, down to individual switches, fibers, racks, etc.

[1] We have other networks that use a more traditional design consisting of large non-SDN routers; our management processes for those networks are somewhat different.

- **Materials ordering**: When we commit to a decision to build or expand a fabric, we take the first (nearly) irreversible step of ordering materials (switches, fibers, etc.). Since these are expensive, after this point we generally avoid changing our intentions. The choice of exactly what materials to order is an automated process driven by MALT models and additional inputs.
- **Fabric deployment**: With the materials on hand, we can build the network (place racks, switches, and SDN controller machines, connect them with fibers, and test that it all works). This step is carried out by humans, using automatically-generated detailed human-readable plans based on the MALT models. We also install the necessary on-switch software using automated processes.
- **Controller and switch configuration**: Once the network is built, we automatically generate SDN controller and switch configurations from the MALT models, with additional configuration inputs.
- **Operations**: Our operators manage the run-time behavior of a network via various tools and services. For example, they might need to "drain" part of a fabric to carry out repairs or upgrades, and then "undrain" it later. The systems that carry out these operational changes are also driven, in part, by MALT models. We also have automated control planes (e.g., [22]) that depend on structural information expressed using MALT.
- **Health and repairs**: We have automated systems to decide whether the network is unhealthy, to diagnose the causes of problems, and to help us understand what systems would be affected by a faulty (or drained) component. These all use MALT as some of their input data.

It should be obvious that we rely on *accurate* and *up-to-date* MALT models for virtually all of our network management automation. The challenges discussed in this paper are all related to preserving that accuracy when our systems are constantly evolving.

2.1 Background on MALT

We briefly summarize the aspects of MALT [27] necessary to understand this paper.

MALT is an *entity-relationship model* representation, not a database. In an entity-relationship model, entities represent "things," which can be abstract (e.g., an entire fat-tree network) or quite detailed (e.g., a specific strand of fiber), or in-between (e.g., an IP adjacency aggregating multiple fibers). Entities have "kinds" (types), names, and attributes, Entities are connected via relationships, which have kinds, but neither names nor attributes.

A collection of entities and relationships forms a MALT *model*. We divide models into *shards*, with some shard-specific metadata. Model shards are typically (but not always) aligned with physical infrastructure boundaries at the city, region, or building scale. Some shards have millions of entities. Please refer to [27, Fig 3.] and [27, Appendix A] for example MALT models. Detailed, machine-readable versions of these examples are available for download [13].

We normally store shards in MALTShop, a purpose-built system that enables easy sharing of models between systems, which depends on naming, access control, and consistency. Shards in MALTShop have names (similar to UNIX pathnames) and access-control lists (ACLs). Every update to a shard creates a new, immutable version, with an immutable version number. MALTShop uses a copy-on-write mechanism to efficiently store many versions of a shard that is being incrementally updated.

MALTShop supports a generic query language, which walks an entity-relationship graph to extract a chosen subset model. It allows one query to span a set of multiple shards.

To manage evolution, each shard can assert compliance with one or more *profiles*. A profile is essentially a contract between a shard's producer and consumers that the shard conforms to a set of predicates. Profiles are versioned; when we need to change how we represent a network, we signal that by creating a new version of the corresponding profile.

Our planners, designers, and operators usually want to think in terms of the high-level abstract designs of these networks (e.g., a Jupiter network is a collection of blocks connected by spine blocks), rather than in terms of specific switches and fibers; the support for multiple abstraction layers in a single MALT model enables this separation of concerns.

2.2 Why automate?

Our work was motivated by our large, heterogeneous networks, but we believe this kind of automated approach would be valuable for a broad range of network operators (although we realize that the market for full-lifecycle automation software may be small).

Automation enables design flexibility and experimentation. The research community has generated a wonderful range of scalable network structures, including Fat-Trees [1], expander graphs [32], F10 [25], etc. These designs exploit path redundancy to support high bandwidth and availability at relatively low cost.

However, most non-hyperscaler enterprises appear not to be using modern multipath networks. In fact, the dominant provider of network hardware recommends a simple three-layer design with large "core" switches at the top, relatively large "aggregation" switches in the middle, and smaller "access" switches (e.g., top-of-rack or ToR switches) at the bottom [8] and other vendors recommend two layers [3].[2] Why do most enterprises avoid multipath networks? Our (admittedly anecdotal) understanding from several experts is that

[2]These are old citations; the age of these documents may reflect an inherent stasis.

most enterprises lack the design tools that they would need to construct and maintain fat-tree networks, let alone less-regular designs such as expander graphs. We speculate that the low adoption rate for research-generated network designs is at least partially due to a lack of tooling, especially to support safe and frequent changes. (There are, of course, other reasons.)

Risk management. Hyperscalers use multipath network topologies [2, 31, 34, 38] also because they support incremental expansion of, and upgrades to, live networks. The need for zero-downtime changes to the structure of these networks is driven by Service Level Objectives (SLOs) targeting 99.99% or better availability, which allows at most a minute of downtime per week. Some providers hope to achieve 99.999% availability, allowing just 5 minutes of downtime per year.

Most enterprise networks are more static than ours; every change creates the risk of a large-scale outage, so operators are extremely change-averse. Their use of non-multipath designs leads to less structural redundancy, which we and other hyperscalers exploit to allow frequent changes at relatively low risk. Our automated tooling both indirectly supports low-risk changes by enabling the use of fat-trees, and directly supports it by allowing us to validate all low-level changes against higher-level intent. Smaller enterprises would benefit from using such automation to mitigate change-management risks; e.g., assigning the same IP address to two different endpoints (a mistake we have made in manual workflows).

Addressable markets. Our work focuses on automated operations on large and frequently-changing networks. Many networks are too small or static to require such automation; how many enterprises actually have large networks? Data on this topic is difficult to find, often because it is only available via high-cost market-research reports. While the number of hyperscalers is small, when we include software-as-a-service and other forms of cloud, there are at least dozens of such providers [29]. The number of hyperscaler data centers is also growing consistently [9]. However, there are many other large non-hyperscaler data centers. There are also thousands of smaller "Points of Presence" (POPs); a typical pattern for both large and small enterprises is to build and maintain small fabrics in many POPs, motivating *frequent* use of design and turnup automation.

Takeaway: Many – perhaps most – network outages result from human error, often associated with physical-network changes [15, 21]; automation with a specific focus on change management can make these changes faster and more reliable. It also enables increased agility and innovation.

2.3 Related work

Prior work on network management has mostly focused on network device configuration management, such as configuration language design [7] and configuration generation for existing networks [5, 24, 34]. Although our networks' configurations are derived from network models, our focus here is on planning for topology designs, and for the physical construction, modification, and eventual decommissioning of networks, as well as for their day-to-day operation.

Most prior work (both academic and commercial) has also typically focused on managing the network as it is now, or on verifying near-term intent (i.e., to be implemented as soon as possible). This includes verification of data-plane [19, 20, 36] and control-plane [4] properties.

In contrast, this paper addresses the challenges of planning for future physical states of a network, especially on how to manage sequences of dependent changes in the face of confounding factors, and on how to validate both individual states and sequences of changes. We note that validated, accurate topology models can enable verification of control-plane and data-plane layers.

Some prior work (e.g., [32, 37]) discussed how topology design affects the complexity of lifecycle management, but did not address how to automate the management processes.

Network operators often expand network topologies to augment capacity [37]. Prior work [38] described how we expand live data center networks, through a layer of patch panels; it uses an integer linear programming algorithm to minimize the number of wires to move, while also maintaining sufficient bandwidth through multiple stages (so as to avoid packet loss). In this paper, we address how those multiple stages are planned and coordinated.

3 Change management challenges

We start by describing some of the many specific challenges in change management.

3.1 Orchestration of physical changes

In traditional, non-automated network management, changes to the physical infrastructure of the network (e.g., adding/moving/removing a switch or link) are typically treated as risky operations. Network operators often limit these to maintenance windows, during which some or all of the network becomes unavailable – and because these windows are disruptive, they can only be scheduled rarely, and then must be carried out as quickly as possible. This approach supports neither rapid evolution nor high availability.

More modern, scalable network designs such as Jellyfish [32] and other expander graphs, Facebook's Fabric [2], and Jupiter [31] use "multipath" designs that exploit path redundancy to support high bandwidth and availability at relatively low cost. Multipath topologies also support incremental expansion of, and other upgrades to, live networks, because their redundancy allows "draining" parts of the network during these operations.

For example, the Jupiter architecture consists of several types of blocks, each of which is a Clos fabric. Some of these blocks ("pods") provide connectivity to racks of machines via Top-of-Rack (ToR) switches; some ("fabric border routers", or FBRs) provide connectivity to WANs and other Jupiters;

some provide connectivity between other blocks in the same Jupiter.[3] This modular architecture allows us to build a large fabric (dozens of blocks) incrementally, rather than paying the capital costs and energy of building an entire fabric before we need all of it. Modularity also allows us to add blocks built from newer (faster/cheaper) switches and links to an existing fabric. In order to manage the evolving connectivity between blocks without having to completely rewire everything, we use a layer of patch panels [38], but even so, adding or removing a block requires significant human effort to reconfigure the patch panels.[4]

When we first started automating operations on Jupiter networks, we ran into the issue that each change to a fabric typically depends on the previous changes. However, our original topology-model representation only allowed us to represent one "intended" view of the network, so we had to serialize changes to a fabric: generate an intended model for one change, then carry out that change, and only then could we generate a model for the next change; the resulting delays become nearly intolerable. Our use of MALT solved that problem, because we could create multiple independent versions of a fabric's model in advance.

However, this still left us with the problem of managing the dependencies between a sequence of models – for example, ensuring that two different changes did not use the same switch port for conflicting purposes. Our solution to this kind of conflict avoidance serializes changes by means of a fabric-level lock, essentially a mutex.

Takeaway: To help us discover and avoid such conflicts as quickly as possible, we have now formalized the relationships between multiple plans for a given fabric using the concept of *PlanPoints*, described in §5, and managed by a service, *TopoPlan* (§5.2).

3.2 Representational change

Our network management systems are highly automated and thus heavily dependent on machine-readable data. This data is primarily represented in MALT, on which we focus in this paper, but we use several other standardized representations, such as OpenConfig [28] for telemetry. These representations must evolve over time, to support novel network designs, new hardware, new management concepts (e.g., failure-independent "zones" for high availability), etc.

For example, initially we did not model connectivity between machines and top-of-rack (ToR) switches, so we did not model machines. However, newer policies for machine-specific security and rate-limiting required authoritative intent for these connections, so we added machines to MALT models.

[3]The original Jupiter design incorporated "spine blocks" to form a folded Clos connecting the pods and FBRs. More recently, we connect those blocks directly without using spine blocks, but sometimes the pods themselves provide transit routing between other blocks [30].

[4]Our more recent deployments replace patch panels with optical circuit switches, which avoid much but not all of the human effort for reconfiguration [30].

Table 1: Acronyms and terms used in this paper.

Acronym/ term	Definition
DCNI	Data Center Network Interconnect
MALT	Multi-Abstraction-Layer Topology representation
MALTShop	A storage system for MALT
MSID	Model-Set ID
MBS	Model-building service
MDS	User-facing design service
MQS	Model-query service
NPI	New-product introduction
Block	Modular unit of fabric design
TopoPlan	Change-management service
UIM	Unified Intent Model
PP	Patch panel
PoR	Plan-of-Record

Consequently, a consumer querying a model for "all devices connected to ToR *T*" now receives not just the connected fabric switches, but also the connected machine entities. In practice, we've found that the complexity and level of detail in our models tends to increase over time.

While we attempt to make most representational changes backward-compatible, this is not always possible; sometimes our best guesses about what matters are wrong. We have learned that seemingly-innocuous changes lead to outages, because of the many clever ways in which programmers accidentally build fragile assumptions into their code.

Since our overall system-of-systems cannot have any significant (multi-minute) downtime, when we need to introduce a new MALT profile (see §2.1) to signal a representational change, we cannot insist that all producers and consumers cut over at the same instant. There are many such systems (especially model consumers) with their own release cycles and constraints on engineering resources. Beyond that, any such change is risky; we would not even want to switch to a new profile without carefully-monitored "canaried" rollouts.

Takeaway: For these velocity and safety reasons, we found it necessary to decouple profile feature *introduction* from model-reader *adoption* of such features. The previous paper on MALT [27] briefly discussed our approach to profile evolution. In §7 and §8.2 we expand that discussion, showing how we use a layer of abstraction to decouple many consumers from the details of profiles.

4 Model-generation systems overview

To help readers understand the implications of managing conflicts between, and changes to, our network plans, we first describe how we generate detailed network models. Appendices §A and §B describe model generation in more detail.

Fig. 2 summarizes the model-generation process, and Table 1 summarizes acronyms and terms used in this paper. First, a model writer (e.g., the network planner) initiates a model change by sending an RPC request ① to a "design service,"

MDS. MDS insulates humans and external systems from a need to understand the details of UIM or TopoPlan (§5.2).

MDS is responsible for (i) translating an imperative user-level request (e.g., "add a new block to a Jupiter fabric") into a sequence of declarative high-level intent changes, and (ii) managing request concurrency (for instance, we preclude certain types of requests from simultaneously modifying the same fabric, and thus sequence those requests with a lock). MDS also collects additional information (e.g., the available IP prefixes) necessary to create the UIM changes.

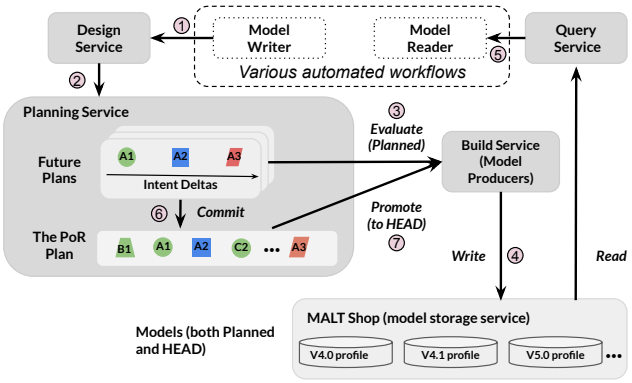


Figure 2: Model-generation systems.

We express model-generation intent in a *Unified Intent Model* (UIM), a form of MALT that abstractly represents the high-level graph of the global network at a given point in time. MDS represents intent changes as deltas to UIM.

This use of abstraction is an improvement on our prior system, in which all network changes consisted of precise orderings of low-level imperative mutations. Those were framed in terms of low-level details, such as the exact type and number of switches and their link-level connectivity; that approach bound decisions too early, making it hard to re-order a sequence of plans. Abstraction helps avoid this early binding.

Once MDS has mapped a request to a sequence of intent changes, it conveys ② this sequence to the TopoPlan plan management service (§5.2). TopoPlan supports parallelism between high-level requests by allowing interleaving of intent changes when they do not conflict. TopoPlan also supports "what-if" analysis, by maintaining multiple (sometimes thousands of) branches of possible future states.

Whenever we need to build concrete MALT models from a UIM plan (e.g., for physical installation), TopoPlan invokes ③ MBS, a "build service" that compiles the high-level intent to MALT models, which are stored ④ in MALTShop. Fig. 2 shows that we simultaneously generate semantically-equivalent concrete models in multiple profiles (see §2.1) – e.g., "V4.0," "V4.1," "V5.0" – to support profile evolution (see §7). For more details on MBS, and many more details on the model-generation process, including examples of UIM, see §A and subsequent appendices.

Models represent the intended network, so mismatches against actual state represent deployment errors (e.g., miscabling, etc.), and we correct reality to match the plans. We use various mechanisms to detect these mismatches, such as neighbor discovery via LLDP (IEEE 802.1AB [17]).

Model readers can query ⑤ these generated models via MQS, a semantic Query Service that also helps support evolution (see §8).

A single high-level operation may invoke these processes multiple times over the course of weeks or months. For instance, when we expand a live Jupiter fabric, we need to do this in several stages, to ensure the network always has sufficient residual capacity during each expansion step. This means we need to generate MALT models for each intermediate stage.

Network model mutations are not real-time. We have safety checkers to block planned changes if they would violate consistency checks, or capacity thresholds designed to leave room for switch or link failures. Systems with real-time goals, such as our SDN controller [11], maintain internal representations of network links, initialized from MALT models.

5 Physical-change plan management

To illustrate the problems of managing concurrent future changes, consider a simple example network with two switches A and B connected via four links (Fig. 3(a)). This network is currently carrying live traffic; hence we call the corresponding MALT representation of the network the "live" model. We would like to expand the network capacity by adding a third switch C (Fig. 3(b)). We call the MALT representation of this future network the "planned" model.

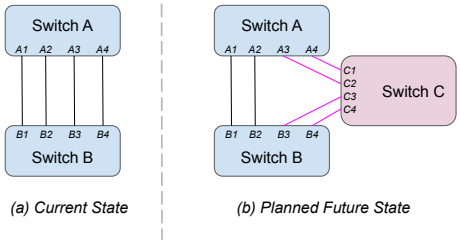


Figure 3: Models for current and planned network.

For several reasons, we want to generate a planned model well in advance. First, it allows us to accurately itemize the physical resources (switches, racks, fiber optic cabling, etc.) required to support switch C. Many of the resources have high costs and lead times: purchasing too much in advance wastes money, while purchasing too little or too late slows our ability to deliver network and compute capacity.

Second, modeling in advance allows us to simulate the future network, and validate it against reliability requirements. *E.g.*, if we must maintain $\geq 75\%$ of normal capacity during expansion, we must change the live network in stages, as shown in Fig. 4. We cannot directly switch between the initial and planned states, which would move two links from switch A (and B) simultaneously, causing 50% capacity loss.

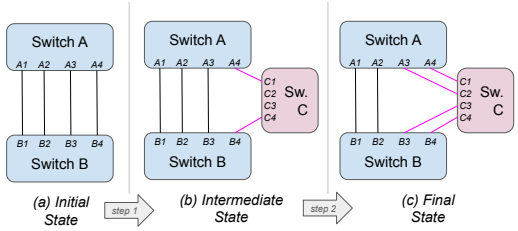


Figure 4: Two-stage migration from initial to final state.

We can also analyze or validate multiple options for a future plan, on metrics such as total cost of ownership (TCO).

5.1 Challenges with concurrent plans

Concurrent management of multiple plans creates several challenges: scaling issues, and (worse) the risks of incorrect planning decisions made because of stale models.

Scaling concurrent plans across a large and changing network: While one could use *ad hoc* methods to manage a set of concurrent plans, such as creating copies of future models in temporary storage, that quickly runs into scaling issues. Because our network is large and frequently changing, we have to manage a large set of concurrent plans. This creates two main problems: sequencing and validation.

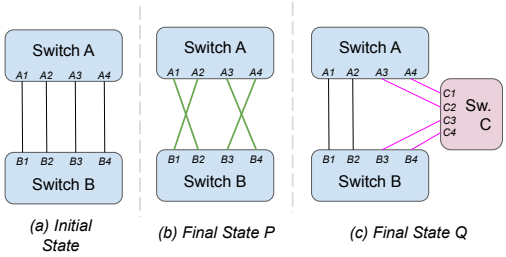


Figure 5: Two conflicting updates, *P* and *Q*.

For example, consider a planned change *P* that rewires the connectivity between switches A and B (Fig. 5(b)). This change conflicts with another planned change *Q* (Fig. 5(c)) to add switch C on ports (A3, A4, B3, B4), and thus we need to decide whether *P* or *Q* should come first.

However, different teams may be making the concurrent changes without coordinating, and the resultant sequence order can affect materials-ordering, instructions for technicians, etc. Worse, the live model is continually changing, and a fabric-level change that is applied to the "live" model (e.g., upgrading one or both switches) may invalidate the preconditions for *P* and *Q*, such as port reservations.

When each fabric has dozens of blocks and hundreds of switches, and must evolve rapidly to meet business needs (via expansions, upgrades, etc.), enforcing serialization on operations such as *P* and *Q* creates painful drag. While manual management of operational concurrency might be practical for just a few fabrics, an enterprise with dozens or hundreds of fabrics would find that unsustainable. Therefore, we need automation-friendly support: for tracking plans, and how they are ordered and sequenced; for rapid conflict-detection and plan-validation; and for speculative analysis of multiple future options (e.g., to cope with supply-chain issues).

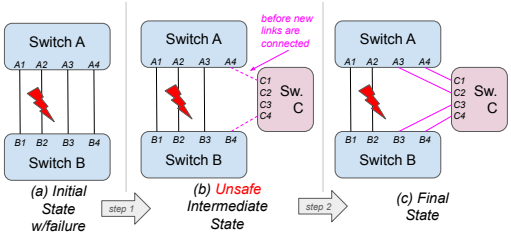


Figure 6: Stale models might lead to an unsafe state.

Stale models: The live network can change unexpectedly, rendering plans stale, along with any derived actions we might have taken. For instance, if a link is removed for repairs (Fig. 6(a)), the initial and final states are both "safe" because they meet our 75% capacity threshold. However, the transition to the intermediate state (Fig. 6(b)) creates an unsafe state (only 50% capacity) while we are changing links. Unless we update and revalidate the models for all intermediate states, we would not detect this risk.

Advance planning can also lead to confusion if, during the long period between plan creation and implementation, constraints change – e.g., supply-chain issues force the use of different hardware.

Thus, given hundreds of potentially-conflicting changes that need to be sequenced, and validated for safety or TCO, manual management of multiple models, or managing placeholder elements in a single "live" model, quickly becomes intractable.

5.2 Plan management service: TopoPlan

To address the many challenges of plan management, we developed TopoPlan, analogous to a software-development version control system (VCS). A network change, specified as patches to high-level intent, may be directly applied to the live (i.e., HEAD) intent. However, the TopoPlan service also allows network changes to be sequenced in branches.

As in a VCS, changes within a branch can be added, removed, reordered, or merged, while branches can be rebased or merged. Branches can represent highly speculative changes (e.g., hypothetical "what-if" scenarios), authoritative changes (e.g., scenarios which we have financially committed to), or changes that are somewhere in between.[5]

Concrete MALT models can be compiled for any change in any branch, allowing us to analyze the network-capacity and TCO implications of any hypothetical future network state. TopoPlan also allows us to detect *conflicts* over limited resources – e.g., two plans trying to use the same switch port for different purposes.

[5] In contrast to traditional VCSes, which are geared toward human users, are text-based, and have change-rate and branching limits, TopoPlan was built with automation in mind, with arbitrary branching, and focuses on sequencing changes to high-level intent, using Protobuf-based intent-patches with special merge semantics [14].

We show concrete examples in §10 of how using TopoPlan greatly improves our deployement and operational efficiency through project pipelining, stacking planned changes, and enabling accurate material orders for future projects.

Plans and PlanPoints: TopoPlan maintains multiple branches of possible future state. A single future change to the network is represented by a *PlanPoint*, which consists of (i) deltas to the UIM, and (ii) a best-estimate timestamp at which the changes will be realized. A series of PlanPoints is called a *Plan*, which groups together a set of network changes into a timeline. A Plan consists specifically of (i) a sequence of PlanPoints, and (ii) a baseline snapshot of MALT models to which the PlanPoint deltas are to be applied. MALT models can be compiled for every PlanPoint in the Plan simply by starting with the UIM of the baseline, and, for each PlanPoint, applying its UIM deltas and compiling concrete MALT models: we term this process *evaluating* a Plan. The compiling process internally invokes the model generation service, as discussed in §4, and attaches the generated concrete models to the PlanPoint.

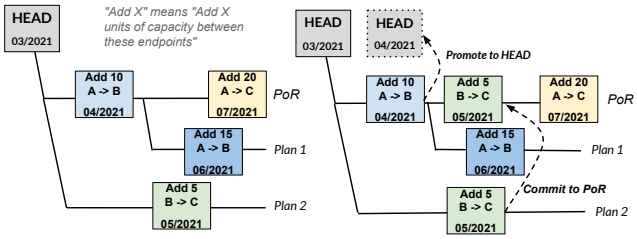


Figure 7: TopoPlan plans (left) and operations (right).

The *Plan-of-Record*, or PoR, is a canonical branch, which stores changes that we have committed to with a high degree of certainty (i.e., we have ordered materials that we really prefer not to waste). In the example of Fig. 7 (left), the PoR contains two PlanPoints that add capacity between B4 sites A -> B and A -> C. Plan 1 proposes increasing the capacity augment between A -> C from 10 to 15 units, while Plan 2 proposes an additional augment between B -> C. Note that, in this example, Plan 1 is "baselined" from a PlanPoint on the PoR, rather than from HEAD.

Committing and promoting plans: Changes on a non-PoR branch can be *committed* (Fig. 2 ⑥) to the PoR (Fig. 7 (right)) if the change does not conflict with other changes on the PoR, and when infrastructure planners sign off on the financial readiness of the change.

Only entire Plans can be committed to the PoR. When a Plan is committed, all its PlanPoints are validated against those on the PoR, and on success they are copied to the PoR. Conflicts are automatically detected by TopoPlan, but manually resolved by backing out, fixing a plan, and retrying. For example, in Fig. 7 (right), attempting the *commit* operation might reveal that we cannot add 5 B -> C units because we have already committed all free ports at C to the 20 A -> C units. We currently rely on human planners to make priority decisions outside TopoPlan, although automating some of these decisions is clearly worthy of future work.

Once the changes in a PlanPoint are ready to be realized in the physical network, the PlanPoint can be *promoted* (Fig. 2 ⑦) to HEAD (the intent representing the current desired state of the network). Promoting a PlanPoint applies its UIM change directly to HEAD, compiles HEAD to concrete MALT models, and removes the PlanPoint from the PoR.

Concrete MALT models can also be generated on demand for any PlanPoint on any branch (including PoR). These models can be used for what-if analyses, for example.

Changing plans: PlanPoints can be added, removed, edited, reordered by changing their timestamps, or merged by collapsing their UIM deltas. Plans can be created, deleted, or rebased by changing their baseline MALT snapshot. Both Plans and PlanPoints are versioned, and their version numbers are incremented on any of theses changes.

Every time HEAD is updated (100s of times per day), TopoPlan rebases the PoR to that new version of HEAD and does light-weight validation to ensure that no highly-certain PlanPoints are invalid. We also perform heavy-weight POR evaluation periodically, but not on every change to HEAD.

We also support backtracking and regeneration of a "known-good state." Thankfully we rarely use this; the complexity of backtracking is sometimes high, especially for plans near their deployment date, since dependencies can force us to unwind multiple changes. Undoing physical changes is expensive and risky, so we use multi-layered validations, as described later in this section, to avoid backtracking.

Lightweight operations: Network changes that are typically planned in advance are expensive capacity-related operations. Most network changes, however, are small-scale local updates (e.g., link repairs, ToR modifications); such changes are typically done immediately and are thus applied directly to HEAD. As a performance optimization, these direct-to-HEAD changes are typically not specified as PlanPoints (and thus skip the PoR), as they almost never affect future planned capacity changes.

Validations: Because compiling concrete MALT models could be slow and must be performed in sequence, we can perform a lighter-weight *UIM validation* operation on a Plan, which computes and validates the UIM for every PlanPoint without full MALT compilation. This runs intent-validation suites for each network and interconnection intent, first separately (to ensure certain properties and assumptions are met), and then globally (to ensure the UIM intents are consistent).

We also periodically run a detailed validation of the concrete models generated from the PoR PlanPoints. This validation is too expensive to run on every commit, so we made a tradeoff between speedy commits and full validation. Full validations still happen often enough to prevent costly mistakes. When our automated validations detect a conflict between plans (e.g., a missed dependency between PlanPoints

that would lead to double-allocation of a port), this usually requires human intervention, to modify one or more plans. Automated resolution is an intriguing research topic.

6 Model generation challenges

Our model-generation system faces several challenges:

Design complexity: Some aspects of our network designs are complex, requiring deep domain knowledge to convert abstract intent into concrete models. In a few cases, e.g., the *Data Center Network Interconnect* (DCNI) layer of connectivity between Jupiter aggregation blocks and spine-blocks, the design is complex enough to require algorithmic support, such as the ILP solver we use to "restripe" the DCNI on changes, while minimizing unnecessary changes to wiring [38].

Heterogeneity: We often add new technologies to our networks – we call these *new product introductions* (NPIs). NPIs sometimes involve novel concepts and so require representational change (see §7). We add NPIs without retiring old products during their useful lifetimes, so our networks are heterogeneous in many design details; our management systems must cope with that. Similarly, we need to be able to rapidly evolve our model-generation system without creating an un-maintainably complex code base.

These challenges, especially our need to support NPIs, pushed us to adopt a layered, modularized design for our model-generation system.

We compose our network designs from fundamental units ("blocks", e.g., server-aggregation blocks and spine-blocks in Jupiter, and B4 blocks [16, 18]). Each block could contain hundreds of chassis and tens of thousands of ports and internal links. A complete data center fabric is composed of up to hundreds of blocks, along with a DCNI. Our fleet has several dozen distinct block types, and each Jupiter or B4 network can have several different generations of blocks.

Our model-generation system uses a modular framework to generate product-specific block-level models, plus additional modules to compose these blocks into a consistent, complete network. For each block type, we have a topology "build unit": a software component that knows how to instantiate that block from high-level intent. These block-level build units are expressed as rules in a concise topology-description language. For many NPIs, we need only create a slightly-modified version of an existing build unit. Other build units, written in traditional programming languages, create inter-block (DCNI) links, assign IP addresses, or validate that the generated models are correct, etc.

When TopoPlan invokes MBS (Fig. 2), MBS creates a dataflow graph, in which the processing steps are the appropriate build units, the input is the intent in UIM, and the outputs are detailed MALT models. MBS constructs this dataflow graph dynamically, to account for changes in our overall network design (the details are beyond this paper's scope).

Scale: Our dataflow graphs are expensive to evaluate, requiring GiBs of I/O and many minutes of CPU time. Concrete MALT models are highly detailed, since they must represent the full underlying detail of our networks. A MALT model representing a single data center network can have millions of entities and relationships, and we have many data centers. Our WAN models have similar scale.

Therefore, MBS uses caching to avoid recomputing previously-generated models. Truly global changes to the Google network are rare and most changes are highly local, so we typically see cache hit rates above 99%, reducing the graph execution costs by two orders of magnitude.

7 Representation evolution

While MALT provides a common, flexible representation, we sometimes need to change its schema, or how we use it, to support NPIs or new management processes.

NPIs generally require changes to model generators, but not always to model readers. For example, a link-speed upgrade from 100G to 200G that otherwise involves no topological changes could be represented by changing the physical_capacity_bps attributes of some EK_PORT entities; this change might not require any updates to model readers.

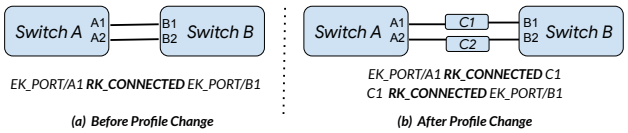


Figure 8: MALT representation of a simple network before and after a profile change.

However, many changes *do* require changes to readers. Fig. 8 illustrates this with a simplified example, where we add new devices (C1 and C2) between switches A and B. Suppose a model reader wants to query what peer switch port is connected to port A1 on switch A. With the old design (Fig. 8(a)), a query could just follow the RK_CONNECTED relationship from A1 to B1. With the new design (Fig. 8(b)), that query would only reach C1, probably not what the query-author intended; the model reader would have to be updated. Because model readers vastly outnumber model writers, such changes are disruptive. Further, a confused model reader (e.g., the configuration generator for our SDN controllers [11]) could cause outages. So, if we fail to realize that a reader needs an update, these changes are also risky.

We mitigate the risk by signalling change through the use of *profiles*. A profile [27] is effectively a versioned contract between model generators and readers, attached to each model. When a generator changes its output in a way that *might* confuse readers or require updates, we increment the profile version. Thus, a reader can detect during testing when it encounters a model with a profile it cannot understand.

MALT profiles by themselves do not avoid the need for updating model readers. Churn due to profile change was a major problem once MALT became widely adopted, which

led us to two requirements: (i) Model readers should have to change their code as little as possible (ideally, not at all) in response to profile changes, (ii) We must provide strong guarantees that migrating to a new profile will not result in regressions to model readers.

Key design choices. To avoid the need to update both model generators and all consumers at the same time, our generators produce multiple semantically-equivalent models, with different profiles, from the same intent (the example in Fig. 2 shows models in versions V4.0, V4.1, and V5.0). This allows some consumers to start testing against a new profile, while most consumers read from the most recent "released" profile (a few stragglers may use older profiles).

To simplify or avoid the code-update problem for most model consumers, we developed a semantic query engine, MQS (§8).

8 Model query service

MALT supports querying models via raw traversal-based queries [27]. However, profile changes could break raw MALT queries (see §7). Generating multiple profile versions mitigates this, but migrating to a new profile version is toilsome: (i) Raw MALT queries are structural rather than semantics-based; (ii) Client code does not always include regression tests for new profiles, and when it does, it is difficult to narrow errors to specific queries; (iii) Client code typically queries for specific data (e.g., ports within a rack), but raw queries return full MALT subgraphs (e.g., all the devices, trays, ports and their relationships), making it hard to predict whether a profile change will affect a given client.

This motivated us to develop MQS, an abstraction layer above MALT query, to minimize profile evolution toil. Stonebraker *et al.* discuss a somewhat different solution to the problem they call "database decay" [33].

8.1 Semantic queries

Instead of writing code that explicitly traverses the MALT graph, as was previously done, developers now write code in a new language that captures the semantics of their query (e.g., "give me all peer ports connected to this device") and hides the mechanics of the MALT graph traversal (e.g., "follow RK_CONNECTED relationships on this device's ports until I reach other ports"). The underlying implementation of these "semantic queries" might be different for each profile; this is hidden from the caller.

Semantic queries are recorded in a registry, allowing us to automatically test that they return the same (or at least, consistent) results across profiles and data sets. This testing framework automatically detects unexpected changes in query results for any potential profile or data change at change-review time, protecting model readers well before such changes can affect production.

8.2 Canned queries

MQS offers a *canned query* API. A canned query is a named function, with defined semantics that are profile-independent. Canned queries are *registered* with MQS. When called to execute a canned query, MQS translates it to an appropriate MALT query, executes it, and processes the returned subgraph to return a set of entities (see Fig. 9).

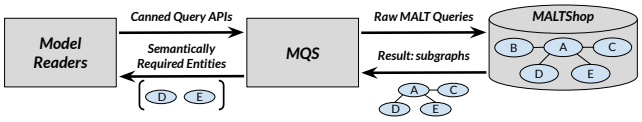


Figure 9: An MQS canned query converts a subgraph to just the entities that are semantically useful to the caller.

Several properties of MQS enable it to return consistent results across multiple profiles, without client code changes: (i) Canned queries can use different MALT queries for different profile versions. (ii) Canned query registration includes semantic tests, allowing centralized regression tests on upcoming "beta" profile versions. (iii) Canned queries return entities with attributes restricted to those that are relevant to clients. This, combined with returning entities rather than MALT subgraphs, greatly reduces the API surface of a model query, allowing easier testing and query evolution.

We run several kinds of centralized validations to ensure minimal profile evolution toil:

(a) **Profile version tests:** we test canned queries across multiple profile versions (including upcoming "beta" versions). Failing tests cause us to either fix the profile or to update the canned-query definition for the new version.

(b) **Data change tests:** When we backport bugfixes or introduce new kinds of products into our network, we introduce changes in multiple profile versions. Before committing these data changes to production models, we generate models in a test environment and compare canned query results between test and production environments.

(c) **Binary and configuration rollout tests:** Prior to releasing a new MQS binary or its associated configuration to production, we run canned queries against the canary and production versions; release automation proceeds only if the results are identical.

9 Cross-shard consistency

We do not try to represent all of Google's networks as one giant model. Not only would the scale create prohibitively expensive memory and communication costs, and complex coordination challenges, but also a single fault could put the entire infrastructure at risk. Instead, we shard the models at natural boundaries (e.g., one shard per datacenter fabric). Sharding allows us to limit the scope of most changes, which greatly simplifies conflict detection.

Many operations span shard boundaries (e.g., adding a WAN link to a datacenter). MALTShop allows queries to trans-

parently span multiple shards, but this raises a question: since we can have multiple future plans (hence, model versions) for each shard, and these plans can be mutated by independent processes, how do we query a *consistent* set of shard versions?

We currently lack a general solution to the shard-consistency problem[6], but we have a workable solution for all datacenter and B4 shards: we (conceptually) rebuild all shards from high-level intent on any intent change (and then use aggressive caching to avoid *actually* building more than necessary). Thus, each intent-change leads to a new *model set*, which is given a unique *model set ID* (MSID) via MALTShop's model-labeling feature. An MSID thus represents a consistent view across these model shards (but unfortunately, not across adjacent shards owned by other systems). MBS uses MSIDs to support consistent queries that cross shard boundaries.

10 Operational experiences

In this section, we dig into several operational experiences and lessons learned. We found the change-management features of TopoPlan to be especially useful in speeding up deployment activities by weeks or months.

10.1 Deadzone reduction

To illustrate one benefit of the PlanPoint abstraction (§5.2), we explain how this speeds up a capacity-delivery process.

We add capacity to Jupiter fabrics, in units of blocks, in a three-step process. **The early modeling** step creates a placeholder version of the block that will exist at a future time. This yields a concrete model from which we can create an order for materials, and it reserves resources, such as patch panel (PP) ports, for this expansion. Once materials are ready for deployment on the data center floor, the **turnup/prepare step** installs and qualifies the new block, but does not connect it to the DCNI. Finally, the **restripe step** gradually folds the new block into the fabric's topology.

However, the same Jupiter fabric may have multiple capacity changes in flight, and we are not always able to overlap the substeps of these augments. Consider the case of a PP expansion followed immediately by a block expansion. We cannot generate the block-to-PP physical striping for the new block until the new PPs and their port reservations are available in a model. Therefore, *if we had only one model*, the first two steps of the block expansion would be blocked until the PP expansion is completed. This results in a *deadzone*, a period when there is work that could be done in a fabric, and is technically unblocked (i.e., we have all the software infrastructure to do the work), but we cannot start because of model-change serialization.

The PlanPoint abstraction allows us to avoid this serialization and effectively pipeline execution. We can create a PlanPoint for the block's early model that uses the post-PP-expansion PlanPoint as its "previous model." The resulting PlanPoint captures a future state when the PP expansion has fully completed and the block's early has been built from it. We can then calculate specific fiber-bundle lengths from this PlanPoint, allowing us to order those bundles long before the PP expansion starts.

[6] We suspect it is similar to distributed-replica consistency, and something like vector clocks might work.

10.2 WAN change management

Our B4 WAN [16, 18] is mutated even more rapidly than a Jupiter fabric, due to its global scale. We change B4 multiple times per day: adding new "neighborhoods," expanding an existing neighborhood, augmenting link capacity between neighborhoods, migrating a neighborhood to a new technology (which entails moving some link endpoints), or removing a neighborhood.

As with Jupiter, we change a live neighborhood or adjacency in multiple steps, to preserve enough capacity to meet SLOs. Thus, we not only generate planned-state models for the end-state topology, but for all intermediate steps as well.

Sometimes projects may be executed independently (e.g., augmenting 2 disjoint edges), but in other cases they are interdependent: e.g., if neighborhood B is port-constrained, adding capacity between neighborhoods A <-> B might first require removing links between B <-> C. Due to real-world constraints, such as supply-chain disruptions, data-center construction delays, etc., the actual execution sequence of these projects rarely follows the global order by which they are initially committed to the plan-of-record (which happens well in advance).

We augmented TopoPlan to express and track such dependencies, adding a layer above TopoPlan to prevent unsupported execution sequences. This allows us to manage WAN projects extending years into the future. Because of the long lead times for materials (fiber bundles, optics, etc.), our ability to pipeline and avoid unnecessary and rigid serialization of plans can shorten deployment timelines by weeks or months.

10.3 Early materials procurement

Supply-chain problems often make procurement of materials (switches, transceivers, etc.) the longest step in a network change. So, we try to order materials as early as possible, ideally before we know exactly how a new network block will fit when we install it (given other in-progress changes with fuzzy completion schedules). In the past, we used heuristics based on prior projects (and some simple scaling rules) to do early orders; this was not always reliable.

Instead, now we speculatively generate a detailed model of the post-change network, and from that we compute and place a precise order. TopoPlan allows us to create these speculative models and manipulate them exactly as we manipulate "real" models, using unmodified software and workflows.

10.4 Software migration

Every software system sooner or later becomes obsolete. Our modeling infrastructure has gone through major software migrations several times, first transitioning consumers from an older representation to MALT, and then changing from a set of monolithic model generators to a modular, more evolvable framework. Managing these migrations without production outages required us to carefully introduce changes in phases. We describe our approach to migration in appendix §C.

11 Summary and future work

We have described how we plan and coordinate changes to large network infrastructures, with a specific emphasis on the need to support parallelism in the face of plan-changes caused by evolving real-world constraints. Since our network-management systems are heavily automated, our plan management must therefore also be automated to keep up with the pace of change, and to avoid mistakes.

We described the TopoPlan system for change management (§5.2), and the fundamental concepts (PlanPoints and branches) it relies on. We discussed how this approach has significantly increased the velocity with which we manage changes to both datacenter networks and WANs (§10.1, §10.2).

Automated network management also depends on explicit models of current and planned topologies, and innovation often requires representational change for our modeling language. We described how we support representational change without major software-engineering disruption, by means of explicit profiles (§7), and a profile-independent query layer that supports many (but not all) model consumers (§8).

Research challenges: While we have already greatly benefited from the work described in this paper, we see many challenges that demand future improvements in change management. In particular, supporting cross-shard consistency, at scale and without funneling all changes through a logically-centralized owner, remains unsolved (see §9). There might also be ways to expand the set of properties that can be validated automatically – not just simple resource conflicts, but also higher-level goals such as fault-resilience.

Acknowledgments

The systems described in this paper represent the design and engineering efforts of many current and former Googlers, in addition to the authors. We would especially like to thank Omid Alipourfard, Drago Goricanec, Xander Lin, Bret McKee, Joon Ong, Sujithra Periasamy, Mitchell Price, Fang Ruan, Michael Rubin, Shouqian Shi, Yiran Su, and Sting Zhou for their contributions.

We thank Vince Muir and Bryan Stiekes for their helpful advice. We also thank, for their feedback on this paper and earlier versions, Aditya Akella, Dennis Fetterly, several sets of anonymous reviewers, and our shepherd, Nik Sultana.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. SIGCOMM*, 2008.

[2] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2014.

[3] Arista Networks, Inc. Arista 7500 Scale-Out Cloud Network Designs. https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7500_Scale_Out_Designs.pdf, 2016.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proc. SIGCOMM*, 2017.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations . In *Proc. SIGCOMM*, 2016.

[6] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proc. SIGPLAN*, 2010.

[7] Xu Chen, Yun Mao, Z Morley Mao, and Jacobus Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Proc. CoNEXT*, 2010.

[8] Cisco. Cisco Data Center Infrastructure 2.5 Design Guide. https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND_2_5a_book/DCInfra_2a.html, 2011.

[9] Kip Compton. Cisco's Global Cloud Index Study: Acceleration of the Multicloud Era. https://blogs.cisco.com/news/acceleration-of-multicloud-era, 2018.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, 2004.

[11] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and

Amin Vahdat. Orion: Google's Software-Defined Networking Control Plane. In *Proc. NSDI*, 2021.

[12] Google. Blaze. http://googleengtools.blogspot.com/2011/06/testing-at-speed-and-scale-ofgoogle.html, 2011.

[13] Google. MALT Example Models. https://github.com/google/malt-example-models, 2023.

[14] Google. Protocol Buffers. https://developers.google.com/protocol-buffers, 2023.

[15] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. In *Proc. SIGCOMM*, 2016.

[16] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN. In *Proc. SIGCOMM*, 2018.

[17] IEEE Standards Association. IEEE 802.1AB-2016: IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery. https://standards.ieee.org/ieee/802.1AB/6047/, 2016.

[18] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proc. SIGCOMM*, 2013.

[19] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. NSDI*, 2012.

[20] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proc. HotSDN*, 2012.

[21] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. The Evolution of Network Configuration: A Tale of Two Campuses. In *Proc. IMC*, 2011.

[22] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proc. SIGCOMM*, 2015.

[23] S. Lee, T. Wong, and H. S. Kim. To Automate or Not to Automate: On the Complexity of Network Configuration. In *Proc. IMC*, 2008.

[24] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. Automatic Life Cycle Management of Network Configurations. In *Proc. SelfDN*, 2018.

[25] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *Proc. NSDI*, 2013.

[26] Guillaume Maudoux and Kim Mens. Correct, Efficient, and Tailored: The Future of Build Systems. *IEEE Software*, 35(2), 2018.

[27] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *Proc. NSDI*, 2020.

[28] OpenConfig Working Group. OpenConfig. https://openconfig.net/, 2014.

[29] Richard Peterson. Top 25 Cloud Computing Service Provider Companies (2021). https://www.guru99.com/cloud-computing-service-provider.html, 2021.

[30] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Connor, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yamsura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking. In *Proc. SIGCOMM*, 2022.

[31] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. SIGCOMM*, 2015.

[32] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *Proc. NSDI*, 2012.

[33] Michael Stonebraker, Dong Deng, and Michael L. Brodie. Database Decay and How to Avoid It. In *Proc. Big Data*, 2016.

[34] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proc. SIGCOMM*, 2016.

[35] Amin Vahdat. NANOG Keynote: Failing Last and Failing Least. https://www.nanog.org/news-stories/nanog-tv/keynotes/keynote-failing-fast-and-failing-least/, 2020.

[36] G.G. Xie, Jibin Zhan, D.A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *Proc. INFOCOM*, 2005.

[37] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding Lifecycle Management Complexity of Datacenter Topologies. In *Proc. NSDI*, 2019.

[38] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C Mogul, and Amin Vahdat. Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks. In *Proc. NSDI*, 2019.

Appendices

Whenever it is necessary to build concrete MALT models from a UIM, we invoke MBS (§A), a "build service" that compiles the high-level intent from TopoPlan to generate MALT models, using product-specific model producers. Collectively, we refer to our module-generating software as Nimble (§B).

A Generic build service: MBS

Model production is the result of the execution of a dataflow graph of *build units* (§B.1). Build units are product-specific, but the orchestration of their execution to construct MALT models is generic. MBS is an execution engine that, given a set of named inputs, (i) constructs the dataflow graph of build units, (ii) executes that graph to produce MALT models, and (iii) transactionally stores the generated models in MALTShop. A set of output models in one transaction is given a *Model Set ID* (MSID), so that model readers can see a consistent snapshot of models.

The main input to MBS is a global UIM representing specifications for the entire Google network at a given point in time; this concisely represents the desired high-level state (e.g., the number of network fabrics in a location, their topology type, interconnect capacity between sites, etc.). MBS is also stateful; build units record their low-level decisions as an input to future model builds, to reduce network churn.

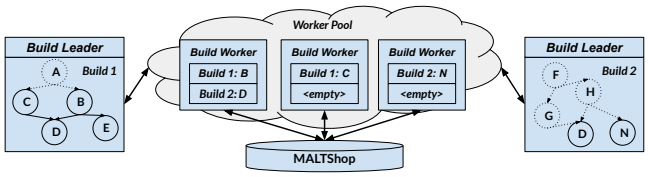


Figure 10: An illustration of our distributed build system showing two independent builds in progress.

Two-phase build. MBS is a distributed dataflow graph evaluator. A graph node can represent either a datum or a rule: data nodes are almost always MALT model fragments, while rule nodes execute build units. Rule nodes are connected to data nodes via either input edges, which specify the input models for a build unit, or output edges, which specify a build unit's output models.

Because the dataflow graph used to generate a set of concrete MALT models is highly data-dependent (e.g., a new MALT model output will be added if we're modeling a new data center location), we dynamically construct this dataflow graph in MBS, by executing a much smaller, static dataflow graph. Special build units in the static graph read the intent model and compute the full, dynamic graph, which MBS then executes to produce concrete models.

Graph execution (static or dynamic) is orchestrated by a leader and worker distributed system (Fig. 10). To execute a graph, MBS assigns it to one leader, which parses and validates the graph, then executes rules in parallel, using a pool of workers, as the rule inputs become available.

Build performance optimization. The resulting full-size dataflow graph, with 100s of thousands of rule nodes, is slow and expensive to evaluate, requiring GiBs of I/O and many minutes of CPU time. Therefore, MBS uses extensive caching to avoid recomputing previously-generated models. Caching is based on hashes of input data nodes and rule specifications, allowing MBS to skip rule evaluation if the corresponding hashes identify a cached output model.

Why we built a distinct system: While Google and others have created extensive distributed dataflow graph execution engines [6, 10] and tooling to efficiently manage and build binaries from source code [12, 26], we created MBS as a distinct system for several reasons:

(a) **Dynamic graphs:** Existing systems expect the execution graph to be provided as an input, rather than itself being dynamically generated based on the input intent models.

(b) **Stateful execution:** Updates to a fabric's concrete model should implement the new planned intent while making as few changes as possible to the existing network (to limit the costs of physical changes). Existing build systems [12] did not easily allow execution i of model generation to consume the output of execution $i-1$.

We suspect the combination of (a) and (b) is novel in the context of build systems.

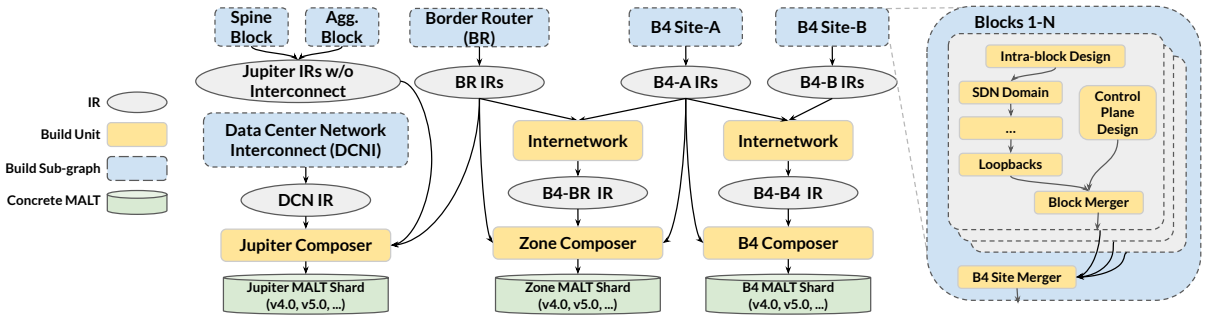


Figure 11: Simplified build graph for producing Jupiter and B4 models, with some areas abstracted for clarity. An expanded view of the B4 subgraph is shown on the right as an example (with some IRs omitted).

B Model generation: Nimble

We developed an ecosystem of re-usable *build unit* software modules, collectively called Nimble, each responsible for constructing a specific part of a model. We organize the build units into a dataflow graph that makes their dependencies explicit. Compared with a monolithic generator, using modular build units respects the knowledge domains involved in designing different aspects of a network (e.g., topology design, IP assignment, port allocation, link striping, etc.).

Modularity also provides natural boundaries to confine problem complexity. For instance, we divide topology design into deterministic descriptor-driven build units and dynamic solver-driven build units. This makes the generation of large-scale topologies with specific properties more tractable than directly solving a monolithic topology optimization problem. Modules without dependencies (or satisfied dependencies) can be executed in parallel, which helps scaling. Finally, modularity supports heterogeneity, since we can add or change just the necessary build units, and reuse others.

Each product's model-producers typically define dozens of build-unit types, to collectively perform the full spectrum of model generation tasks for each fabric and interconnect in the network, including topology generation, IP address allocation, SDN controller domain assignment, etc. We roughly categorize build units into four classes: (i) intra-block topology generators (§B.1.1), (ii) inter-block capacity generators (§B.1.2), (iii) model fragment composers (§B.1.3), and (iv) model validators (§B.1.4). While the first two categories both generate topologies, their designs are drastically different. The model fragment composers and validators are at the end of the design pipeline, and are necessary to support model sharding and ensure correctness.

Fig. 11 depicts a simplified example build graph involving the model producers for both B4 and Jupiter networks. Build units shown in the figure are abstracted, simplified representations of those we use in production.

B.1 Build unit overview

A build run creates two categories of model data: (i) the final concrete MALT model(s) adhering to a specific profile, and (ii) Intermediate Results (IRs): internal data, also represented in MALT, that facilitate communication between build units, but are not exposed for external consumption.

Some IRs are ephemeral, since they are consumed by downstream build units in the same build run. Some IRs are memoized results, required for the future build runs (e.g., the set of allocated ports). Memoization avoids the need to always build all output from scratch. Some IRs are part of the inputs to the composers (see §B.1.3), which process and stitch these together to generate the concrete, versioned model-shard outputs. Each build unit may take as input the model intent, previous IRs, and previous concrete models, and outputs IRs or concrete models.

We compose our network fabrics from fundamental units called "blocks" (e.g., server-aggregation blocks and spine-blocks in Jupiter, and B4 blocks [16, 18]). Each block could contain hundreds of chassis and tens of thousands of ports and internal links. A complete data center fabric is composed of up to hundreds of blocks, along with a "Data Center Network Interconnect" (DCNI). Our fleet can have several dozen distinct block types, as their technology evolves.

Each block type has a fixed, deterministic internal topology, but the DCNI or WAN interconnect depends on their dynamic properties (e.g., block type, uplink capacity, port availability, etc.). We generate each intra-block topology from a declarative topology description (§B.1.1) but we generate the DCNI and WAN interconnects via solvers that optimize link striping and port allocations (§B.1.2).

B.1.1 Intra-block topology generator

The intra-block topology generator is effectively a compiler that parses a *topology descriptor* that declaratively describes the desired block topology, and emits a corresponding MALT fragment of the detailed design. These descriptors are parameterized templates for each topology type. This process is deterministic, and does not require a complex solver, given the regular design of block internals.

The descriptor language expresses intent for a given block type as a hierarchy of modules, with specific entity-kinds (e.g., packet switches or ports) as leaf nodes. Descriptors can

specify entity attributes, which can be parameterized (e.g., the index of a module within its parent), and naming schemes. Interconnection patterns between entities, such as ports, are selected via *mappers* such as *fullmesh* or *biject*, within a scope called a *group*. When invoking the compiler, a build unit can pass certain parameters (abstracted from the model intent) to the descriptor; e.g., to control the number of racks within the B4 block. This allows one descriptor to support a variety of topologies for the same block-type generation. (§B.2 provides more details on the descriptor-based approach.)

While the build-graph structure is flexible, for simplicity, the typical pattern in most model producers' pipelines is for the first build unit to invoke the compiler to construct the backbone IR for a block, while subsequent build units build on this IR with additional entities and relationships (e.g., allocating management IPs, SDN control domains, etc.).

B.1.2 Inter-block capacity design

The internal topology of a block is typically static throughout its lifecycle. Inter-block connectivity, however, is frequently updated as we add or decommission blocks, add capacity between blocks, or fix incorrectly-wired fibers. Updates to the topology must meet capacity and availability requirements, and also minimize change to deployed reality (i.e., not move fibers unnecessarily). We have several solver-based build units for inter-block connectivity that tackle different classes of problems. *E.g.*, the *Internetwork* build unit in Fig. 11 uses a generic interconnect design and management solver for block-level striping, port-allocation, interface IP addressing, etc. This is used to generate WAN connections between B4 sites, and to the *Border Routers* of the data center fabrics.

These build units try to maximize the path diversity between pairs of sites or blocks, which improves tolerance of physical faults (e.g., link-, chassis-, block-level failures), while adhering to physical deployment constraints (e.g., minimizing the number of wasted ports).

For pairs of B4 sites, for example, each site may span several Points of Presence (PoPs), each containing multiple blocks; the interconnect solver minimizes the maximum imbalance in block-to-block, PoP-to-PoP, and block-to-PoP allocations of links across block-pair. We formulate this as a mixed integer programming optimization problem.

The design problem for the data center network interconnect (DCNI) has a large optimization space. We discuss that solver in appendix §B.3.

B.1.3 IR composers

At the end of each model generation run, a set of *composers* is responsible for stitching together the IRs produced by the upstream built units to create the concrete model shards. The MALT models are sharded for a variety of reasons, such as domain isolation and scalability, as discussed in [27]. We have a dedicated composer for each model shard.

Within each shard, the composer processes and merges IRs, based on their tagged profiles, to generate profile-compliant models for all supported profile versions. We define our pipeline such that any profile-agnostic processing (e.g., resource allocation, etc.) is done as early as possible, while profile-dependent modeling is typically branched further downstream, at or near the composers; this helps ensure data consistency across profiles.

B.1.4 Validators

During each model generation run, we also validate attributes, and design rules in several categories: (i) *Intent validation* ensures that the UIM is internally consistent and its changes are legitimate; e.g., the UIM satisfies the properties required by model producers. (ii) *Property validation* focuses on validating network-specific invariants we expect in each model (e.g., ports do not conflict, IP addresses are not duplicated), and (iii) *intent-to-model validation*, which is designed to harden the intent-to-model translation that typically requires dynamic solvers (e.g., whether the striping between a B4 neighborhood and Jupiter delivers the intended capacity, while satisfying diversity and balance requirements). Finally, (iv) *model-change scope validation* ensures the scope of model changes matches the corresponding change in the intent. [7]

During the course of development, all these validation suites have caught some exceptions, especially for NPIs, which if left undetected would have caused network outages or costly deployment errors.

B.2 Details: intra-block topology generator

This section provides additional details on how we support a high-level approach to block-level design. The intra-block topology generator includes (i) topology descriptors that fully declare the topology and (ii) a compiler that parses the descriptors and translates them into MALT models. These descriptors deterministically declare the intended topology.

We explain several key aspects of topology descriptors using an example snippet (Fig. 12) of the descriptor of a B4 Stargate block [16]. We simplified the descriptor for clarity.

Hierarchy: A network is a hierarchy of *modules* with a single tree root and multiple branches. The root module of a block is usually an EK_NETWORK, and its name is globally unique.

Modules: A module, the basic building block in descriptors, defines one MALT entity kind and the topology within the entity. As our networks are heterogeneous, multiple modules (with distinct names) may refer to the same entity kind. For instance, we have two module definitions for EK_PHYSICAL_CHASSIS in Fig. 12. A descriptor can have multiple instances for the same module.

[7] MBS also performs more basic validations such as MALT lint checks and profile schema checks.

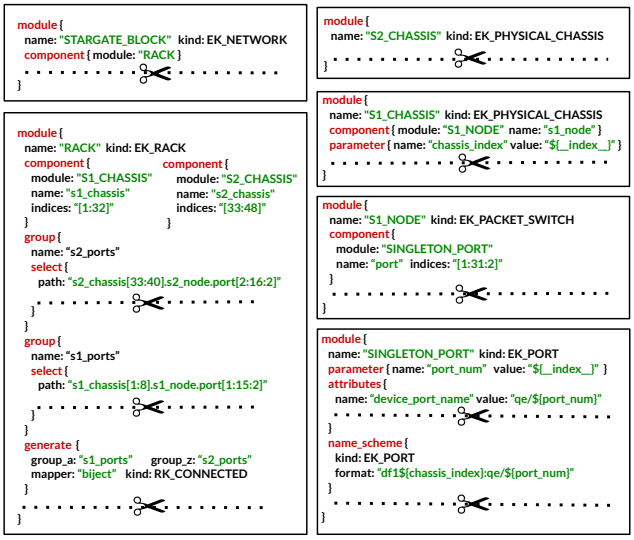


Figure 12: A snippet of a topology descriptor for a Stargate Block.

Components: The topology within a module is defined by recursively including other modules as its *components*. The number of components (of the same entity kind) included in the parent module is concisely expressed using indices. For instance, in the module "S1_NODE", the singleton port component is defined with indices [1:31:2], indicating that there are 16 entities in this module, with indices from $\{1, 3, ..., 31\}$. The default relationship between a module and its components is that the module entity RK_CONTAINS all its components. Other relationship types can be specified to override that default, at the component type granularity.

Attributes: Entities in MALT models can have attributes, such as taxonomy (e.g., chassis type) and state (e.g., link is in turnup). Attributes specified in topology descriptors are self-contained: i.e., they are either static values, or they are deterministically computable using the *parameters* defined within the upstream hierarchy (branch) of the entity.

Parameterization: The descriptor is not another topology programming language – we omitted constructs such as conditionals. However, allowing basic parameterization of modules offers useful flexibility and concision. The most common use of parameters is to pass information top-down. For instance, the chassis_index parameter of the "S1_CHASSIS" module is subsequently used by the singleton port contained by the chassis. If a module is componentized with indices, we have multiple instances of this module; the "${__index__}" provides each instance's index. Parameters defined in a module are recursively visible to all components and sub-components in the module.

Relationships: To create relationships between components or create intra-block links between ports, the descriptor introduces the *Group* operation to *Select* a set of components within a module hierarchy, and then applies a *Generate* operation to generate relationships or links between the components of the two groups. A *mapper* is used to decide how the components in *group_a* are mapped to those of *group_z*. Two common mappers are *biject* (pairwise, requiring the two groups to have the same number of items) and *fullmesh*. In Fig. 12, the "RACK" module uses group and generate to define how S1 ports and S2 ports are connected.

Naming schemes: Each MALT entity has a unique ID, combining its name and entity kind. A topology descriptor specifies a naming scheme by either constructing it ad-hoc (potentially using parameters) or simply referring to a preconstructed regular expression (in most cases).

Given a descriptor, the topology compiler is responsible for parsing the descriptor and generating the corresponding MALT model fragment. Internally, the compiler parses modules top-down, builds multiple branches based on the component indices while enforcing parameter scopes within each branch, and finally constructs entity names, attributes and relationships. The compiler also allows customized mappers in the Generate operations to compensate for topology irregularities. Because the compiler does not make any topology-specific assumptions, it is generic and reusable across all descriptors.

For most model producers, their first build unit instructs the topology compiler to construct the backbone IR for a network. Subsequent build units decorate the IR with additional entities and relationships (e.g., allocating management IPs and SDN control domains). Fig. 11 depicts these data flows. When invoking the compiler, a build unit can pass certain parameters (abstracted from the model intent) to the descriptor; for instance, to control the number of racks within the B4 block. This enables us to support a variety of topologies for the same network generation while reusing the same descriptor.

B.3 Details: inter-block topology generator

This section expands on the discussion in §B.1.1, describing how we generate models for the Data Center Network Interconnect (DCNI) in a Jupiter network.

DCNI overview. A Jupiter network uses a layer of Patch Panels (PPs) between server blocks and spine blocks. Each block (server or spine) directly connects to the front side of PPs, and the block-to-block connectivity is (indirectly) established by cross-connecting the back side of PPs. Having a PP layer removes a lot of complexity that would be introduced by directly connecting server and spine blocks, e.g., reduced fiber length and human labor. This is discussed in detail in [38].

We use the term DCNI (Data Center Network Interconnect) to describe the two collections of "physical" links, i.e., block-to-PP links and PP cross-links. The challenge for inter-block design is to produce an optimal DCNI. We call the resulting block-to-block paths "logical" links.

Block-to-PP generation. The Jupiter model producer has a dedicated DCNI build unit. This build unit first performs the block-to-PP link generation to construct the physical topology, and then generates PP cross-links to produce the desired

logical block-to-block topology. For a given block, its block-to-PP fibers fan out equally across every patch panel using a predetermined pattern (i.e., agnostic to intent). Once the block is deployed, such fibers never change.

The block-to-PP links are dynamically allocated in three steps: (i) A *block-to-PP spec generator* translates the UIM into an IR specifying the number of PP ports needed for each block; (ii) A *patch panel port allocator* takes that IR as input, and dynamically assigns available ports to block-to-PP fibers in an on-demand manner. It must read the previous models in order to honor deployed reality; (iii) A *bad port swapper* reads bad-port UIM, and uses reserved ports to replace those bad ports. An external device-repair workflow automatically creates bad-port UIM to record faulty ports. Since block-to-PP link restripe does not affect traffic (because these new links have not been used to carry traffic), this restripe is accomplished in one shot, i.e., without phasing.

PP cross-link generation. Given the physical topology, obtaining the desired logical topology is a complex problem. Thus, the DCNI build unit invokes a dedicated external solver, which translates PP cross-link generation into an ILP (Integer Linear Programming) problem, as described in [38].

Although the ILP solver is able to compute the desired final state, it does not naturally support the crucial requirement that the DCNI must carry live traffic during restripe. This requirement is addressed by having multiple incremental restripe stages, where each stage only alters a small portion of topology, to ensure that the network has sufficient residual capacity in all stages. We use an automated *expansion planner* to decide the number of required stages. Given a restripe request, the expansion planner iteratively searches for the smallest number of stages C (starting from 1) that satisfies the residual capacity requirement. For each evaluated value of C, the DCNI build unit invokes MDS to generate a series of C hypothetical models that resemble each intermediate stage. The residual bandwidth is then calculated, from these models, by counting the number of added/removed links. We provide additional details for the restripe process in §B.4.

Another goal of the DCNI build unit is to ensure topology stability. By its nature, the logical topology solver is not deterministic. Invoking the solver in different model generation runs could cause the DCNI to change arbitrarily, even without any intent change, forcing us to do useless re-wiring. Thus, we use a persistent memoization layer, allowing the solver to store its calculated topology solutions persistently. This both prevents redundant calculation, and ensures that the DCNI build unit generates deterministic output.

B.4 Case study: Jupiter restripe

When we add blocks or spine blocks to a Jupiter network, we must perform a "restripe" operation, to redistribute uplinks from aggregation blocks to spine blocks. When we do a restripe that could affect in-service links, we must do it incrementally, to avoid distrupting too much capacity at once.

In this section, we use the restripe process to illustrate details about the DCNI build unit.

There are three major catogeories of Jupiter data center restripes. (i) Front-only restripe: the restripe only adds/removes the block-to-PP links that do not have fiber jumpers on the back side, so it touches only the front side of patch panels. These added/removed links do not carry live traffic, so incremental restripe is not required. (ii) Back-only restripe: the restripe only alters the PP cross-links, so the scope of change is limited to the fiber jumpers on the back side. Incremental restripe is required since it changes the logical block-to-block links. (iii) Combined restripe: the restripe alters both block-to-PP and PP cross-links at the same time. It is the most labor-intensive process compared with the other categories, and it is also required to be an incremental restripe.

We observed that for most common restripe use cases, the model update process could be divided into one front-only restripe and multiple stages of back-only restripe. As an example, when a new block is added to a Jupiter fabric, we first add the block-to-PP links (front-only restripe), and then we use multiple incremental stages to update PP cross-links (back-only restripes). Such granularization helps reduce the sequence requirement in the workflow, and allows better parallelism among different workflows that are expanding different parts of a Jupiter fabric.

Intermediate restripe stages. Similar to most model changes, the DCNI design is also driven by intent changes. Fig. 13 shows an example of high-level Jupiter data center intent (on the left side), which is part of the global UIM. The fields that are related to DCNI are highlighted in blue (e.g., the number of PP chassis). The DCNI build unit is responsible for translating the Jupiter intent into DCNIShape, a protocol buffer defined as the input to the topology solver interfaced with the *PP-Cross-Link Gen* build unit, a subcomponent of the logical DCNI build unit.

The PP cross-link restripe process consists of small stages to gradually transform the current topology to the desired one given by the solver. We use a single PP rack as the smallest granularity of restriping stages, because it provides natural grouping of the logical block-to-block links. Thus the whole restripe process boils down to rewiring patch panels in multiple incremental stages.

Conceptually, the model for each stage could be summarized as a 4-tuple: $(\mathsf{T}_o, \mathsf{PP}_o, \mathsf{T}_n, \mathsf{PP}_n)$, where T_o and T_n denote the two sets of all physical block-to-block links in the old and new topologies; PP_o and PP_n form a partition of all PP indices, where PP_n represents PPs that are restriped to have links from T_n, and PP_o represents the rest of patch panels that still have links from T_o. Fig. 14 shows a 3-stage restripe with three PPs:

(a) Before restripe. There is no old topology. The "new" physical topology T_0 connects server block 1 and spine block 1 via PP 1, 2, and 3. Thus the tuple is

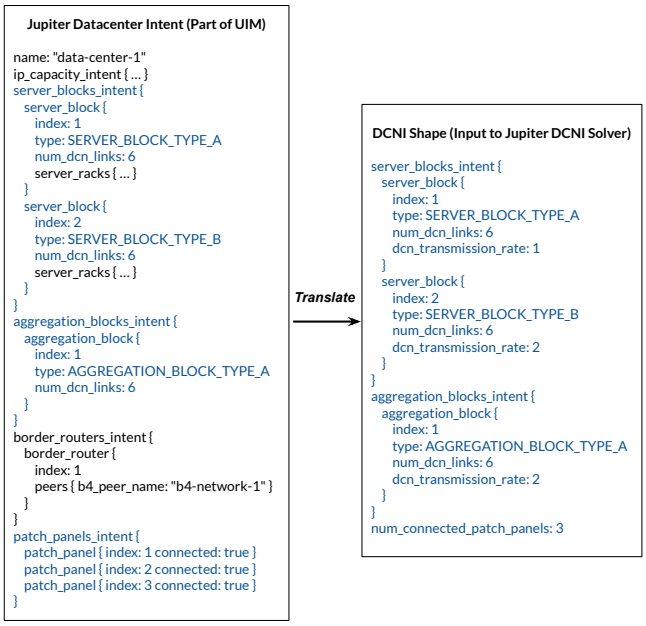


Figure 13: An example of DCNI intent translated from UIM.

$(\{\},\{\},\mathsf{T}_0,\{1,2,3\})$.

(b) Stage #1. The new topology physical T_1 connects server block 1, 2, and spine block 1. Only patch panel 1 has been updated. Thus the tuple is $(\mathsf{T}_0,\{2,3\},\mathsf{T}_1,\{1\})$.

(c) Stage #2. The new topology T1 remains the same. PP 2 is further folded into the logical topology. Thus, the tuple could be summarized as $(\mathsf{T}_0,\{3\},\mathsf{T}_1,\{1,2\})$.

(d) Stage #3. The new topology T1 remains the same. All PPs are updated into the logical topology. There is no more "old" topology as restripe is completed. Thus, the is tuple $(\{\},\{\},\mathsf{T}_1,\{1,2,3\})$.

This tuple is stored in an IR called MaskedDcnTopology IR. The PP-Cross-Link Gen build unit will read the previous MaskedDcnTopology IR and Jupiter intent to update the tuple, and then translate the tuple to an intermediate topology.

C Live migration to new infrastructure

Our legacy modeling infrastructure[8] had numerous problems, including scaling issues, and weak support for schema evolution and parallel operations. For production safety, we could not simply stop using the old systems and immediately migrate its many users to our new model-generation infrastructure (Nimble) until we were confident that the old and new systems were functionally equivalent. Even small discrepancies can cause serious outages. However, we could not just stop using the old systems while we tested the new ones, as that would have blocked all network operations for weeks or months.

[8]The legacy infrastructure was similar to a relational database schema.

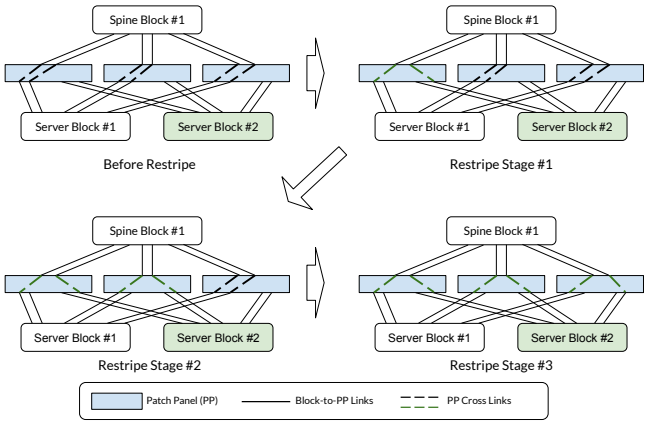


Figure 14: An example of 3-stage restripe with three PPs.

We conducted a live migration from the old systems in four phases:

Phase I: Exporter. We wrote an *exporter* pipeline that converted legacy models to equivalent MALT models. This allowed most model consumers to migrate to MALT. At this point, the legacy models were still treated as authoritative.

Phase II: Validation. To avoid production outages, we had to ensure that models produced by Nimble were functionally equivalent to the exporter-generated models. We built a pipeline that reverse-engineered UIM and relevant state from the exporter's output, yielding intent that we could feed to Nimble. We could then check that Nimble and the exporter generated identical MALT models from semantically-equivalent intent.

Phase III: Read migration. After the model equivalence checks passed consistently for several weeks, we atomically flipped the MALTShop paths where the exporter and Nimble wrote their models, automatically causing readers to consume Nimble-generated models. We staged this changeover on a per-model-shard basis, enabling read migration for some shards while we were still fixing differences for other shards.

Phase IV: Write migration. We then migrated all model writers (e.g., capacity-delivery and data center expansion workflows) to use MDS. After this, we deprecated the legacy model-design tools.

Phased migration turned out to be invaluable. Together with our high-level design principles for reliable systems [35], we managed to finish the fleet-wide migration of our critical modeling infrastructure without any production incidents.

AAsclepius: Monitoring, Diagnosing, and Detouring at the Internet Peering Edge

Kaicheng Yang[†,§], Yuanpeng Li[†,§], Sheng Long[†,¶], Tong Yang[†,§], Ruijie Miao[†], Yikai Zhao[†], Chaoyang Ji[¶], Penghui Mi[¶], Guodong Yang[¶], Qiong Xie[¶], Hao Wang[¶], Yinhua Wang[¶], Bo Deng[¶], Zhiqiang Liao[¶], Chengqiang Huang[¶], Yongqiang Yang[¶], Xiang Huang[¶], Wei Sun[¶], Xiaoping Zhu[¶]

[†]National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University
[§]Peng Cheng Laboratory, Shenzhen, China [¶]Huawei Cloud Computing Technologies Co., Ltd., China

Abstract

[1] Network faults occur frequently in the Internet. From the perspective of cloud service providers, network faults can be classified into three categories: cloud faults, client faults, and middle faults. This paper mainly focuses on middle faults. To minimize the harm of middle faults, we build a fully automatic system in Huawei Cloud, namely AAsclepius, which consists of a monitoring subsystem, a diagnosing subsystem, and a detouring subsystem. Through the collaboration of the three subsystems, AAsclepius monitors network faults, diagnoses network faults, and detours the traffic to circumvent middle faults at the Internet peering edge. The key innovation of AAsclepius is to identify the fault direction with a novel technique, namely PathDebugging. AAsclepius has been running for two years stable, protecting Huawei Cloud from major accidents in 2021 and 2022. Our evaluation on three major points of presence in December 2021 shows that AAsclepius can efficiently and safely detour the traffic to circumvent outbound faults within a few minutes.

1 Introduction

Network faults, including congestion, link failures, BGP misconfigurations, *etc.*, occur frequently in the Internet [1–8]. Obviously, network faults could degrade the network performance, and even lead to outages [9, 10], threatening the connectivity of the Internet. As a cloud service provider (CSP) which serves users at the Internet peering edge, the quality of service (QoS) for users is significantly harmed by the frequent network faults.

As pointed by BlameIt [11], from the perspective of CSPs, network faults can be classified into three categories based on where they occur (see Figure 1): 1) *cloud faults* which occur in the *cloud network* (cloud AS [2]); 2) *client faults* which occur in the *client network* (client AS); 3) *middle faults* which occur in the *middle network* (all AS'es between the cloud AS and the client AS). First, the cloud network is fully controlled by CSPs. We have already deployed a battle-tested system in Huawei Cloud to heal cloud faults, and this paper does not focus on cloud faults. Second, the client network is neither controlled by nor directly connected to CSPs. Therefore, when client faults occur, what we can do is to request the corresponding Internet service providers (ISPs) [11] for fault healing. Third, the middle network is not controlled by but directly connected to CSPs. Same as other CSPs, Huawei Cloud is connected to the middle network through dozens of points of presence[3] (PoPs). All PoPs and datacenters in Huawei Cloud are interconnected by a private backbone. Leveraging the private backbone, the traffic between clients and Huawei Cloud can go through different PoPs. This implies that we can choose different middle paths (paths in the middle network) by choosing different PoPs, so as to handle middle faults. In

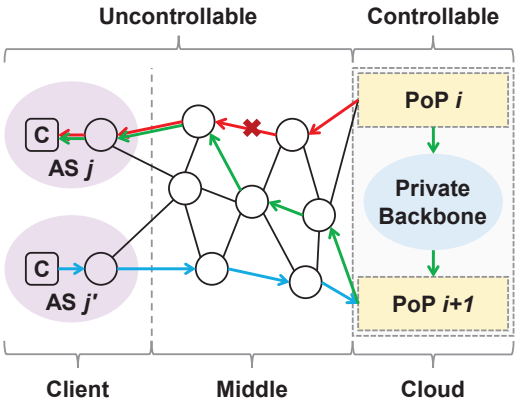


Figure 1: Three categories of network faults. "C" refers to a client in the client AS, and "×" refers to a middle fault. When a middle fault occurs in the red path, we can detour the traffic to egress at PoP_{i+1}, so as to route the traffic along the green path to circumvent the fault.

[1] Co-primary authors: Kaicheng Yang and Yuanpeng Li. Corresponding author: Tong Yang (yangtongemail@gmail.com).

[2] AS refers to autonomous system, which is a very large network or group of networks with a single routing policy.

[3] Point of presence is the point where Huawei Cloud accesses the Internet, see more details in Section 2.1.

summary, this paper focuses on middle faults, and identifies client faults as well.

The design goal of this paper is to design a fully automatic system to minimize the harm of middle faults. The system should have three main functions: fault monitoring, fault diagnosis, and traffic detouring. First, the system should persistently monitor the middle and client networks in a lightweight manner to detect and report faults. Second, the system should accurately diagnose the reported faults at fine granularity. Third, for a middle fault, the system should efficiently and safely detour the traffic to a healthy path, and detour the traffic back immediately when the fault disappears.

Towards the design goal, the most important and challenging issue is to identify the fault direction, since no existing works handle this problem. Suppose we find a middle fault between a PoP_i and a client AS_j. If a fault occurs in the path from PoP_i to AS_j, we define the direction of this fault as outbound direction, and similarly we define inbound direction. In this case, fortunately, we have many PoPs, and thus outbound faults can be quickly circumvented: detouring the traffic through another PoP_{i+1} to the client (see Figure 1). This action is quick because it only needs to update the routing table of PoP_i. If a fault occurs in the inbound direction from AS_j to PoP_i, unfortunately available solutions to change the traffic path need to change the routing tables of all routers in the middle network, which is obviously slow. The above observations pose great importance on *identifying the fault direction.* Further, it is challenging to identify the fault direction. On the one hand, the middle network where faults occur is completely out of our control. On the other hand, our monitoring results do not include direction information.

Researchers and engineers have proposed various solutions for network faults at Internet scale [1, 11–24]. Among them, BlameIt [11], Edge Fabric [12], Espresso [13], Entact [14], and CPR [15] are most related to our application scenarios. However, as shown in Table 1, these works only have two of the three main functions, and none of them identifies the fault direction. BlameIt monitors and diagnoses network faults in the cloud environment. However, it does not support traffic detouring and fault direction identification. Edge Fabric, Espresso, Entact, and CPR support traffic detouring, but they do not diagnose network faults. In summary, all existing works do not meet our design goal.

Aiming at the design goal, we design a system, namely **AAsclepius** (**A**uto**Asclepius**). AAsclepius consists of three subsystems: a monitoring subsystem, a diagnosing subsystem, and a detouring subsystem. Each subsystem is responsible for implementing a main function. Below we only show how the diagnosing subsystem addresses the above challenge of identifying fault direction.

Diagnosing subsystem: AAsclepius uses a decision tree with intuitive design to achieve the **accuracy** and **fine granularity** of diagnosis. Our experienced experts have spent a long time configuring the thresholds (§ 5) used in the decision tree. These thresholds have proven to work excellently after two years of validation. Our key innovation is to propose a technique, namely *PathDebugging*, to address the most important and challenging issue, *i.e.*, identifying the fault direction. For a middle fault between a PoP_i and a client AS_j, the idea of PathDebugging is to replace the path from AS_j to PoP_i with a zero-fault path. In spite of the simple idea, the implementation procedure is rather complicated, and the details are provided in Section 5.3.

To date, AAsclepius has been running for two years, keeping Huawei Cloud free of major accidents. In December 2021, we conducted an evaluation on three major PoPs. The results show that for all outbound faults, AAsclepius can efficiently and safely detour the traffic to circumvent them within a few minutes.

2 Settings

2.1 Cloud Infrastructure

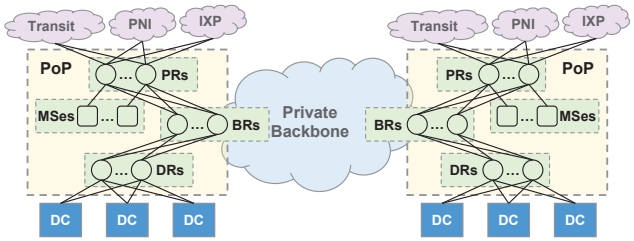


Figure 2: A PoP houses multiple peering routers (PRs) to access the Internet, backbone routers (BRs) to access the private backbone, and datacenter routers (DRs) to access datacenters. Each peering router is connected to a monitoring server (MS) cluster (currently we only deploy one server per PR) used for monitoring QoS (§ 4.2).

In order to serve approximately one billion geo-distributed users, Huawei has deployed a cloud consisting of dozens of PoPs and datacenters globally, as well as a private backbone that interconnects all PoPs and datacenters. To access the Internet, as shown in Figure 2, each PoP houses multiple peering routers (PRs) as edge routers, exchanging BGP routes and traffic with transit providers, private network interconnects (PNIs), and Internet exchange points (IXPs) outside the PoP. The interconnectivity between all PoPs and datacenters provided by the private backbone greatly improves the flexibility of traffic detouring. Traffic from any datacenter is first routed to the backbone routers (BRs) via datacenter routers (DRs). Through the private backbone, the traffic can then be routed to egress at any PoP. Similarly, the inbound traffic can also ingress at any PoP. Such flexibility of traffic detouring is the basis of AAsclepius (§ 6).

To serve users, each PoP announces a distinct set of IP prefixes as its *dominating prefixes*. Obviously, the PoP at which the inbound traffic will be routed to ingress is only determined

Desired functions	AAsclepius	BlameIt [11]	EdgeFabric [12]	Espresso [13]	Entact [14]	CPR [15]
Fault monitoring	✓	✓	✓	✓	✓	✓
Fault diagnosis	✓	✓	×	×	×	×
Direction identification	✓	×	×	×	×	×
Traffic detouring	✓	×	✓	✓	✓	✓

Table 1: Comparison with prior solutions.

by its IP destination address. Unfortunately, if a PoP fails to announce its dominating prefixes, the inbound traffic with IP destination address in those prefixes will not be routed to Huawei Cloud. To achieve fault resilience, each PoP additionally announces the dominating prefixes of the other PoPs with multiple duplicate AS'es prepended to their `AS_path`[4] (*e.g.*, 12345, 12345, 12345). In this configuration, the traffic is normally routed as before. If a PoP fails to announce its dominating prefixes, after BGP converges, the traffic that originally ingresses at that PoP will be routed to ingress at another PoP. Therefore, the availability of Huawei Cloud service is guaranteed as long as one PoP remains operational.

We have built a traffic monitoring system in Huawei Cloud, passively counting the sizes of flows entering or exiting Huawei Cloud at <Source IP, Destination IP> granularity for billing using programmable switches. This system not only helps AAsclepius with fault monitoring as it can determine the AS'es and IP /24 prefixes which contain clients of Huawei Cloud, but also helps AAsclepius with traffic detouring as it can provide visibility to the traffic volume between any PoP and AS. While it is cost-efficient to passively monitor stateless traffic volume, it is too expensive to maintain per-flow state for a large cloud to monitor network faults, and therefore AAsclepius still uses *active probing*.

2.2 Domestic Network Infrastructure

China's network is mainly controlled by three top-tier transit-providers. To serve geo-distributed users, each top-tier transit provider builds its own large-scale backbone network with sufficient intra-bandwidth interconnecting with all its client networks. By comparison, the inter-bandwidth across different transit providers is usually limited. Such infrastructure guarantees that from a PoP's perspective, the IPs in the same AS share similar middle paths. It also motivates AAsclepius' design (further shown in § 6): for traffic suffering from network faults, AAsclepius detours it across different PoPs within the same transit-provider for better performance, instead of detouring it across different transit-providers in the same PoP.

3 AAsclepius Overview

Currently, because most traffic of Huawei Cloud exits from PoPs deployed in our country, we have deployed AAsclepius on a large scale in our domestic infrastructure. As mentioned above, AAsclepius has three subsystems: a monitoring subsystem, a diagnosing subsystem, and a detouring subsystem.

[4]The BGP AS path attribute sequentially lists the AS numbers comprising the path to the destination.

Below we briefly present the modules and workflow of these subsystems (see Figure 3). Because most of Huawei Cloud traffic is still IPv4 traffic, AAsclepius targets at only IPv4 traffic currently. The terminologies frequently used in this paper are shown in Table 2.

Table 2: Terminologies frequently used in this paper.

Terminology	Meaning
Active IP address	An IP address which will respond to ICMP probes. (§ 4.1)
IP*	A representative IP address in a prefix for probing. (§ 4.1)
Victim-AS	An AS which is identified as suffering from faults. (§ 4.3)
Fake fault	A network anomaly that causes a healthy-AS to be identified as a victim-AS. (§ 5.1)
Backup PoP	PoP_{i+1} is the backup PoP of PoP_i. (§ 5.3)
Victim traffic	The traffic that is suffering from faults. (§ 6)

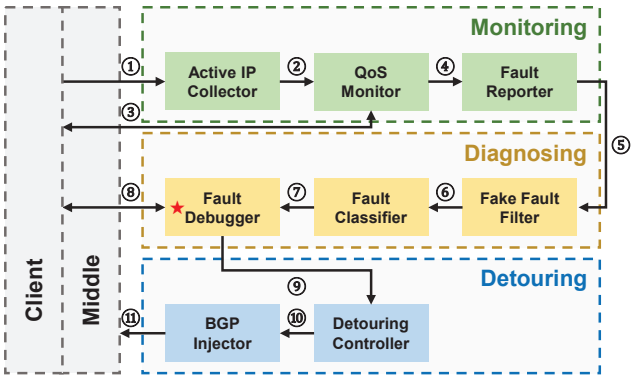


Figure 3: The workflow of AAsclepius.

Monitoring subsystem (Figure 3 top). This subsystem consists of three modules: an *active IP collector*, a *QoS monitor*, and a *fault reporter*. The QoS monitor and the fault reporter are deployed on a per-PoP basis.

① The active IP collector monitors all IP /24 prefixes in the AS'es which contain clients of Huawei Cloud, and collects all active IP addresses.

② The active IP collector selects representative IP addresses for each IP /24 prefix and informs the QoS monitor.

③ The QoS monitor in each PoP receives representative IP addresses, and runs QoS agents to send ICMP probes to representative IP addresses in order to measure their packet loss rates.

④ The QoS monitor aggregates packet loss rates at AS-granularity, and passes them to the fault reporter.

⑤ The fault reporter in each PoP receives the aggregated packet loss data from the QoS monitor in the same PoP, then identifies and reports victim-AS'es to the diagnosing subsystem.

Diagnosing subsystem (Figure 3 middle). This subsystem consists of three modules: a *fake fault filter*, a *fault classifier*, and a *fault debugger*. The fake fault filter is deployed on a per-PoP basis.

⑥ The fake fault filter receives reported victim-AS'es from the fault reporter in the same PoP, identifies and filters fake faults, and outputs the other faults to the fault classifier.

⑦ The fault classifier receives the faults from the fake fault filter in many PoPs, classifies the faults into three categories, and passes middle faults to the fault debugger.

⑧ The fault debugger receives middle faults from the fault classifier, and runs debugging agents to monitor a second path for each middle fault.

⑨ The fault debugger then compares the monitored packet loss rates of QoS agents and debugging agents to identify the direction for each middle fault.

Detouring subsystem (Figure 3 bottom). The detouring subsystem consists of two modules: a *detouring controller* and a *BGP injector*. The BGP injector is deployed on a per-PoP basis.

⑩ The detouring controller receives middle faults from the fault debugger, generates detouring strategy for each middle fault, and outputs the strategy to the BGP injector.

⑪ The BGP injector receives detouring strategy from the detouring controller, generates corresponding BGP routes, and announces them to PRs or DRs to detour the victim traffic.

4 Monitoring Network Faults

The monitoring subsystem consists of the active IP collector (§ 4.1), the QoS monitor (§ 4.2), and the fault reporter (§ 4.3). Each module will be introduced in one subsection.

4.1 Collector: Selecting Representative IPs

The first module in the monitoring subsystem is the active IP collector (collector for short). The collector, deployed in a VM in Huawei Cloud, is used to collect and select active IP addresses. It proceeds in two steps. 1) For only IP /24 prefixes in the AS'es which contain clients of Huawei Cloud, we decide to actively send ICMP probes to IP addresses in them, and regard the monitored performance as the QoS for users in this prefix. To guarantee the accuracy of active monitoring, we need to select active IP addresses as targets for probing. Therefore, the **first step** is to collect active IP addresses from the AS'es which contain clients of Huawei Cloud. 2) Considering that the active IP addresses in each IP /24 prefix share similar middle paths and client paths[5], it is of no value to monitor all of them. To achieve lightweight monitoring, we need to further reduce the overhead by selecting representative IP addresses (IP*s for short) in each prefix for probing. Therefore, the **second step** is to select IP*s in each IP /24 prefix, and then passes them to the second module, *i.e.*, the QoS monitor. Below, we describe the two steps of the collector in detail.

The first step of the collector proceeds as follows. The collector maintains *health points* as the indicator of activeness for each IP address, and only scans IP /24 prefixes in the AS'es which contain clients of Huawei Cloud by sending ICMP probes to each IP address in them periodically. In each round of scanning, if an IP address responds to the ICMP probes, its health points will increase; otherwise, its health points will decrease. The health points will decay over time, so as to indicate recent health status. After each round of scanning, the collector selects the IP addresses whose health points exceed a predefined threshold as active IP addresses.

The second step of the collector proceeds as follows. After each round of scanning, for each IP prefix, the collector sorts the active IP addresses in it based on the health points from the highest to the lowest. The IP addresses that usually belong to gateways (*e.g.*, .1, .254) will be given bonus health points before sorting. Then, for each IP /24 prefix, the collector selects the top-k active IP addresses in it as IP*s, and passes them to the QoS monitor. By default, we set k to 5. For those prefixes that contain clients of Huawei Cloud, we set a larger k to better reflect the QoS. Note that the information of the AS'es and IP /24 prefixes containing clients of Huawei Cloud is provided by the traffic monitoring system built in Huawei Cloud (§ 2.1).

4.2 Monitor: Monitoring QoS

The second module in the monitoring subsystem is the QoS monitor (monitor for short). The monitor is used to monitor the QoS for users in different AS'es. It is deployed based on the following considerations. 1) As Huawei Cloud can serve users from any PoP, we need to monitor the QoS by monitoring the performance of IP*s from every PoP. 2) To avoid the disturbance of cloud faults and traffic detouring (§ 6), we should start monitoring as close to the PR as possible. 3) Considering that the cloud traffic can egress at any PR, we should monitor all PRs to achieve full coverage of monitoring, *i.e.*, send probes to each IP* through all PRs. Therefore, AAsclepius deploys the monitor on a per-PoP basis as follows. As shown in Figure 2, for each PR, AAsclepius deploys a server directly connected to it, namely *monitoring server*. On each monitoring server, the monitor runs a QoS agent to send ICMP probes, which will egress at the PR connected to the monitoring server, so as to achieve full coverage. This deployment can easily scale out from a monitoring server to a cluster consisting of multiple monitoring servers, so as to improve the capability of the monitor.

The monitor proceeds in two steps. First, the monitor monitors the performance of each IP* received from the active IP collector. Specifically, **every minute**, the monitor executes a round of monitoring: for each IP*, each QoS agent sends ICMP probes to it, and computes its average packet loss rate

[5] Client paths refer to the paths in the client AS.

among all QoS agents, so as to achieve full coverage. We call this process **QoS monitoring**.

Second, the monitor aggregates the performance of IP*s at AS-granularity, and then passes the aggregated performance data to the third module, *i.e.*, the fault reporter. Specifically, after each round of monitoring, for each AS, the monitor computes the average packet loss rate of all IP*s in it as its packet loss rate. The reason behind is as follows. It is expected that we can monitor the QoS in every IP /24 prefix to achieve fault monitoring at prefix-granularity, but not all IP /24 prefixes contain active IP address. The network infrastructure in our country can ensure that the IP*s in the same AS share similar middle paths, and thus share similar middle faults (discussed in Section 2.2). Therefore, monitoring the QoS at AS-granularity will not degrade the sensitivity to middle faults. In our deployment, we find the above attribute also applies to the network of every ISP in every province. Therefore, we call the network of every ISP in every province a *pseudo AS*, and aggregate the performance of IP*s at pseudo-AS-granularity to ease maintenance. *In the rest of this paper, we always use AS to refer to pseudo AS.*

4.3 Reporter: Reporting Victim-AS'es

The third module in the monitoring subsystem is the fault reporter (reporter for short). The reporter, deployed on a per-PoP basis, is used to identify victim-AS'es and filter victim-AS'es suffering from transient faults. The reporter proceeds in two steps.

In the **first step**, the reporter receives the aggregated performance data from the QoS monitor deployed in the same PoP, and identifies whether an AS is suffering from faults based on three observations. The first observation is that the packet loss rate of each AS remains relatively stable when no fault occurs; once a fault occurs, the packet loss rates of the AS'es suffering from the fault suddenly increase. Therefore, for each AS, we decide to monitor the variation of its packet loss rate, and use a *fault threshold* to identify whether it is suffering from faults. The second observation is that different AS'es have different patterns of packet loss rates, because different AS'es point to different middle paths and client paths, and are maintained by different ISPs. Therefore, we should use an AS-specific fault threshold rather than a unified one. The third observation is that the condition of the Internet varies over time, and therefore the fault threshold should dynamically evolve as time goes by.

Based on the above considerations, the first step proceeds as follows. The reporter considers that each AS is in *healthy-state*, *i.e.*, not *victim-state* (suffering from faults), at the beginning. Here, we use "victim" instead of "faulty" to avoid confusion, as a client AS suffering from faults may have no faults occurring in its own network (the faults can occur in the middle path between the AS and the PoP to affect the traffic). The reporter maintains packet loss rates for each AS in recent W minutes. For AS_j, the reporter computes the average ($l_{avg}(j,t)$) and standard deviation ($\sigma(j,t)$) of packet loss rates, to characterize its current pattern, where t (Minute) is the current time. Let $\mathcal{T}_v(j,t)$ be the fault threshold used to identify whether AS_j is suffering from faults at time t. We set

$$\mathcal{T}_v(j,t) = l_{avg}(j,t-1) + \max(\tau\sigma(j,t-1), \delta)$$

where δ is a constant (by default 3%) to filter minor faults. For AS_j, every minute, the reporter receives its current packet loss rate $l_{cur}(j,t)$ from the QoS monitor, and then compares $l_{cur}(j,t)$ with $\mathcal{T}_v(j,t)$. If $l_{cur}(j,t)$ exceeds $\mathcal{T}_v(j,t)$, the reporter identifies that AS_j is suffering from faults, and transits AS_j to victim-state. We call an AS in victim-state as a *victim-AS*, and a time period in which an AS is continuously in victim-state a *victim-period*. For a victim-AS, to avoid the disturbance of high packet loss rate caused by faults, its fault threshold will stop to update when it transits to victim-state from healthy-state. When the packet loss rate of a victim-AS stays below its fault threshold for W minutes, we consider that all its packet loss rates in the sliding window are not affected by faults. In this case, the victim-AS will transit back to healthy-state and restart to update its fault threshold. In our deployment, we set the window size W to 60 and τ to 3. Here, we provide some insights into their settings. We set window size W to 60 (1 hour) because it can stably present the recent condition of Internet. As shown in Figure 10, the average packet loss rate varies smoothly per hour (with less than 0.1% variation). We set τ to 3 because *3σ-rule* is widely used in outlier detection. When there is no fault, the loss rate of each IP* can be regarded as independent and identically distributed. Therefore, the average loss rate of IP*s ($l_{avg}(j,t)$) follows normal distribution according to central limit theorem, and applies to 3σ-rule.

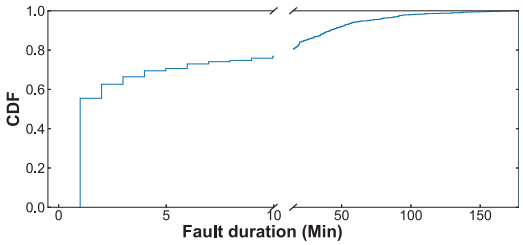


Figure 4: CDF of fault duration of victim-AS'es.

In the **second step**, based on another observation, the reporter filters victim-AS'es suffering from transient faults, and reports the remaining victim-AS'es to the diagnosing subsystem. The observation is that transient faults disappear quickly, and thus have limited harm to our cloud service. Therefore, we would like to ignore transient faults. Based on the above consideration, the second step proceeds as follows. Suppose the current time is t. The reporter only reports the victim-AS'es that satisfy the following two requirements to the diagnosing subsystem: 1) the victim-AS is identified as suffering from faults at time t; 2) the victim-AS has ever been identified as suffering from faults for at least M minutes continuously in the current victim-period. In our deployment, we set M to

3. We select M based on an analysis of the distribution of the fault duration[6] of victim-AS'es in three major PoPs in December 2021. As shown in Figure 4, the fault duration of almost 2/3 of victim-AS'es is below 3 minutes. Therefore, setting M to 3 can efficiently filter transient faults without compromising much timeliness.

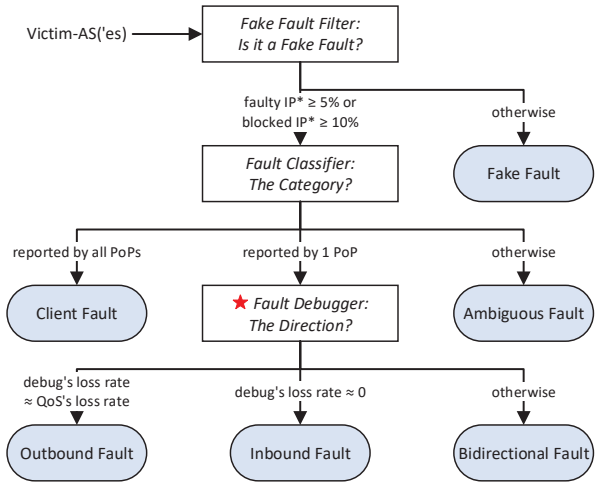


Figure 5: Decision tree.

5 Diagnosing Network Faults

In this section, we show how to use a decision tree in the diagnosing subsystem to achieve accurate and fine-grained diagnosis. As shown in Figure 5, there are three critical decision nodes in the decision tree: the fake fault filter (§ 5.1), the fault classifier (§ 5.2), and the fault debugger (§ 5.3). The design of the structure of the decision tree is quite intuitive. With the reported victim-AS'es, first, the fake fault filter first filters those fake faults that lead to misreported victim-AS'es. Second, the fault classifier pick out the middle faults that AAsclepius may circumvent from the true faults. Third, the fault debugger classifies the middle faults into inbound/outbound/bidirectional faults to guide the subsequent traffic detouring. Each decision node will be introduced in one subsection.

5.1 Fake-filter: Filtering Fake Faults

Motivation: The fault reporter in the monitoring subsystem reports a victim-AS when the average packet loss rate of all IP*s in the AS increase. However, the increase of average packet loss rate does not mean that a real fault occurs. For example, when a router that hosts an IP* is updating its operating system, it may not be able to respond to ICMP probes. Thus the packet loss rate of this IP* will suddenly increase to 100%, which also leads to the increase of average packet loss rate. If a victim-AS is actually healthy, we say it is suffering from **fake faults**. Therefore, it is desired for each PoP to filter all victim-AS'es with fake faults.

[6]The fault duration of a victim-AS refers to the interval between the first time and the last time it is identified as suffering from faults in the same victim-period.

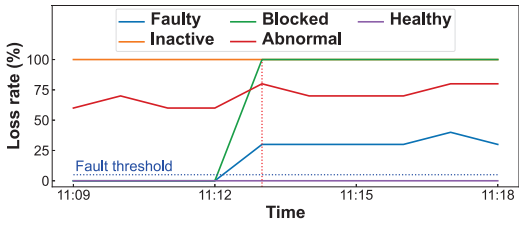


Figure 6: IP* Classification.

Workflow: To prevent fake faults from interfering with our diagnosis, AAsclepius deploys the first module in this subsystem: a fake fault filter for each PoP (*Fake-filter* for short). In each PoP, Fake-filter performs analysis for each IP* to identify whether a reported victim-AS is caused by a fake fault, and we will only diagnose real faults in next two modules. The input of Fake-filter includes: a victim-AS$_j$ with its IP*s, and the historical loss rate of each IP*. Then we analyze why the average packet loss rate increases a lot. Fake-filter divides the IP*s into 5 categories according to their loss rate (see Figure 6), but only use three categories (faulty IP*s, blocked IP*s, and healthy IP*s) to identify fake faults. The 5 categories of IP*s are detailed below.

- Symbols: Suppose at time t_{fault}, victim-AS$_j$ transits to victim-state, lasts for $M = 3$ minutes, and then is reported to Fake-filter. Note that the setting of M is discussed in Section 4.3. Let $T_{pre}[3Min]$ be the 3 minutes before t_{fault}, and let $T_{post}[3min]$ be the 3 minutes after t_{fault}.
- **Faulty IP*s:** We call an IP* a faulty IP* if it satisfies the three conditions: 1) its average loss rate in $T_{pre}[3Min]$ is lower than fault threshold $\mathcal{T}_v(j,t_{fault})$; 2) its average loss rate in $T_{post}[3Min]$ is in $[\mathcal{T}_v(j,t_{fault}),100\%)$; 3) its highest loss rate in $T_{post}[3min]$ is lower than 100%. We consider that faulty IP*s are affected by the fault occurring at t_{fault}.
- **Blocked IP*s:** We call an IP* a blocked IP* if it satisfies the three conditions: 1) its average loss rate in $T_{pre}[3Min]$ is lower than $\mathcal{T}_v(j,t_{fault})$; 2) its average loss rate in $T_{post}[3Min]$ is higher than $\mathcal{T}_v(j,t_{fault})$; 3) its highest loss rate in $T_{post}[3Min]$ reaches 100%. These IP*s are often blocked in two cases. First, they are added into a blocklist by some network devices (IP blocking). Second, with a small probability, they suffer from serious faults.
- **Healthy IP*s:** We call an IP* a healthy IP* if its average loss rates in both $T_{pre}[3Min]$ and $T_{post}[3Min]$ are lower than $\mathcal{T}_v(j,t_{fault})$. We consider that healthy IP*s are not affected by the fault occurring at t_{fault}.
- **Inactive IP*s:** We call an IP* an inactive IP* if its average loss rate in $T_{pre}[3Min]$ is 100%. Inactive IP*s previously responded to ICMP probes, but stop responses before $T_{pre}[3Min]$.
- **Abnormal IP*s:** We call an IP* an abnormal IP* if its average loss rate in $T_{pre}[3Min]$ is in $[\mathcal{T}_v(j,t_{fault}),100\%)$. We suspect that abnormal IP*s have suffered from other network anomalies or network faults occurring before $T_{pre}[3Min]$.

Identifying fake faults: After classifying faults into the above five categories, we analyze whether it is a fake fault. Obviously, the last two categories cannot be used for identification. We define two metrics using the first three categories. We define *faulty IP* ratio* as $\frac{\#\text{ faulty IP*s}}{\#\text{ total}}$, and define *blocked IP* ratio* as $\frac{\#\text{ blocked IP*s}}{\#\text{ total}}$, where # total = # faulty IP*s + # blocked IP*s + # healthy IP*s. We set two thresholds for the two metrics respectively. According to long time maintenance, we find when If the faulty IP* ratio is no less than 5%, or the blocked IP* ratio is no less than 10%, Fake-filter reports the fault as real; otherwise, Fake-filter reports the fault as fake.

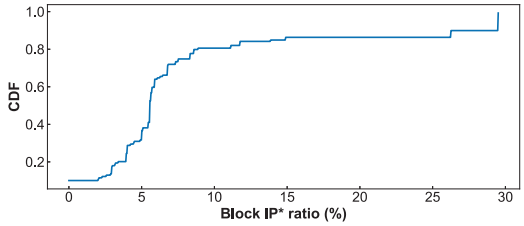


Figure 7: CDF of block IP* ratio of victim-AS'es without faulty IP*s.

While the threshold for faulty IP* ratio is mainly set according to long-time operational experience, we provide some insights in the setting of the threshold for block IP* ratio. When real faults occur, an affected IP* will be either faulty IP* or block IP*. When there is no real fault, the faulty IP* ratio keeps close to 0, but the blocked IP* ratio often reaches a little larger than 0 (*e.g.*, 5%) due to IP blocking. Therefore, we should set a larger threshold for blocked IP* ratio to filter the fake faults caused by IP blocking. We analyze the distribution of the block IP* ratio of all reported victim-AS'es without faulty IP*s in May 10th, 2023. These reported faults are of high probability to be fake faults as no faulty IP* is reported, and a fault with a larger block IP* ratio should have a larger probability of being real fault. As shown in Figure 7, about 80% reported victim-AS'es have less than 10% block IP* ratio. Therefore, setting the threshold for blocked IP* ratio to 10% may efficiently filter most fake faults while not missing serious faults.

5.2 Classifier: Identifying Fault Category

Overview: To identify fault category, AAsclepius deploys the second module in this subsystem: the fault classifier (classifier for short). Recalling in the previous module, Fake-filter in each PoP filters fake faults, and reports the other faults to the classifier. The input is the received faults from many PoPs in every minute: <PoP$_1$,victim-AS list$_1$> ... <PoP$_n$,victim-AS list$_n$>. For each victim-AS, the classifier outputs its fault category: client fault, middle fault, or *ambiguous fault*. For each middle fault, the classifier will report the corresponding PoP and victim-AS to the third module, *i.e.*, the fault debugger.

Workflow: Our observation is the same as the prior work [11]: most of the time the victim-AS is caused by either client faults or middle faults. We use the number of PoPs that report each victim-AS to identify fault category, and there are three cases.

- Case 1: If a victim-AS is reported from only one PoP, the classifier identifies the fault as a middle fault.
- Case 2: If a victim-AS is reported from all PoPs, the classifier identifies the fault as a client fault.
- Case 3: Otherwise, it is too difficult to identify, and we have to concede, and the classifier identifies the fault as an ambiguous fault.

The reason of the above classification is as follows. Considering a client in an AS, for different PoPs, the middle paths are different, while the client paths are usually similar. Therefore, 1) that a victim-AS is reported from 1 PoP is incurred by middle faults with high probability; 2) that a victim-AS is reported from all PoPs is incurred by client faults with high probability; 3) that a victim-AS is reported from multiple but not all PoPs, and the above probabilities of middle faults and client faults both decrease a lot. Our maintenance results (see Figure 8) show that the probability of case 3 is around 7%, and thus currently we just ignore case 3.

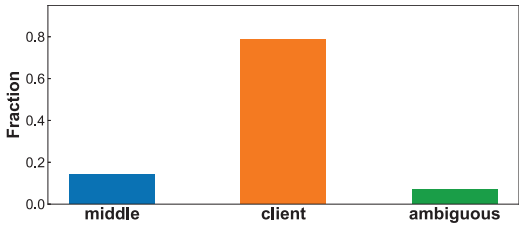


Figure 8: Fraction of each category of faults.

Operational experience: In our daily maintenance, we find a situation that will degrade the accuracy of our diagnosis: although multiple PoPs report the same victim-AS, the time they first report is not always the same, but sometimes with a one-minute or two-minute difference. A potential reason behind is as follows. In the early stage of a network fault, there may be only a small amount of congested links. Due to the randomness of the routing inside the victim-AS, the ICMP probes may go through paths with slight difference, and thus the results measured at different PoP points may be slightly different. For example, suggest that ICMP probes sent from both PoP$_i$ and PoP$_{i+1}$ reach a router using equal cost multi-path strategy (ECMP) in a client AS, and there are two equal-cost paths towards the destination. One of the path first becomes congested due to the randomness of hash functions, while the other one remains uncongested. Suggest that the ICMP probes from PoP$_i$ are forwarded to the congested link, and those from PoP$_{i+1}$ are forwarded to the uncongested link, then only PoP$_i$ reports this AS as a victim-AS. As the congestion further evolves (which may take one or two minutes), both PoP$_i$ and PoP$_{i+1}$ report this AS as a victim-AS. In this case, if the classifier diagnoses a victim-AS once it is reported, the victim-AS may be misdiagnosed because there could be some PoPs that have not reported this victim-AS in

time. To address this issue, for each reported victim-AS, we decide to delay the diagnosis for two minutes. Specifically, only when a victim-AS is continuously reported from a PoP for three minutes, the classifier starts to diagnose it. The classifier diagnoses the victim-AS based on the number of PoPs that report it in the past three minutes, instead of the number in the current minute.

5.3 Debugger: Identifying Fault Direction

Overview: Recalling that the previous module reports a PoP and a victim-AS to this module. Suggest PoP_i is reported to find that victim-AS is suffering from middle faults. This module, namely fault debugger (debugger for short), is used to further identify and output the direction of the middle fault. There are three faults with different directions: outbound faults, inbound faults, and bidirectional faults. We propose a novel technique, namely *PathDebugging*, to perform the debugging process. This technique is the key novelty of AAsclepius.

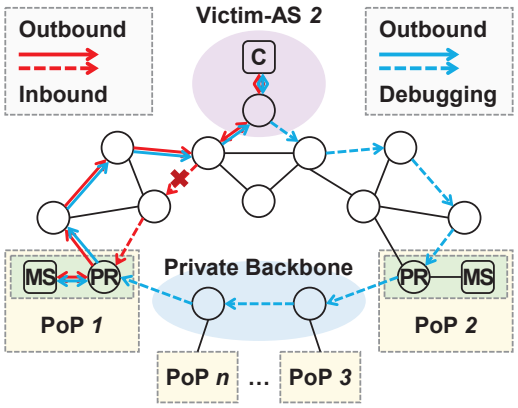


Figure 9: PathDebugging. "**MS**" refers to the monitor server in the PoP, "**C**" refers to a client in the victim-AS_2, and "×" refers to a middle fault.

Statement of fault direction: As shown in Figure 9, there are four paths between the monitor server (MS) and the client (C): two outbound paths (solid lines with arrows), one inbound path (dash red lines with arrows), and one debugging path (dash blue lines with arrows including the part across the private backbone). In this Figure, the reported PoP and victim-AS from the previous module are PoP_1 and AS_2. This means that we only know there is a fault between PoP_1 and AS_2, but do not know the fault direction. Outbound faults, inbound faults, and bidirectional faults point to different directions. For different fault directions, we will use different detouring strategies in the detouring subsystem (§ 6).

Rationale: To identify whether the fault is in the outbound path or the inbound path, the idea of our key technique PathDebugging is to replace inbound path with a zero-fault path, which is named the **debugging path**. After replacement, we monitor the packet loss rate of the ICMP probes between the reported PoP and victim-AS: 1) if the loss rate does not change, it means it is an outbound fault; 2) if the loss rate decreases to near 0, it is an inbound fault; 3) if the loss rate decreases but not reach 0, it is a bidirectional fault. Next we show how to set a debugging path and route the ICMP reply packets along the debugging path.

Workflow: Recall that each PoP has multiple PRs (peering routers), each PR is connected to a monitoring server, and each monitoring server runs a QoS agent. We deploy another agent named *debugging agent* in each monitoring server. These two agents are very similar except that they use different source IP addresses. By leveraging the debugging agent and BGP prefix announcement, next we show how to set the debugging path and let the ICMP reply packets follow the debugging path.

Phase 1: announcing BGP prefixes. We have deployed many PoPs: PoP_1, PoP_2, ..., PoP_n. We associate every two adjacent PoPs for debugging and detouring. We call PoP_{i+1} the *backup* of PoP_i. For PoP_i, we assign a unique IP /24 prefix to the debugging agents in it, and the prefix is announced by all PRs in the backup PoP_{i+1}. For example in Figure 9, the PRs in PoP_2 announce the unique prefix of the debugging agent in PoP_1.

Phase 2: activating the debugging path. This phase aims to let the ICMP reply packets follow the debugging path, and observe the packet loss rate. Take Figure 9 as an example. The debugging agent in PoP_1 chooses a source IP address from its unique IP /24 prefix Pre_1, and then sends ICMP packets along the outbound path (the solid line) to the client. The ICMP reply packets will follow the debugging path (the dash blue line crossing the private backbone), because the PRs in PoP_2 announce the unique prefix Pre_1. Note that there is no fault in the debugging path for the following two reasons. First, as the middle fault is identified when only PoP_1 reports the fault, there must be no fault in the path from the victim-AS to PoP_2. Second, the private backbone is adequately provisioned, and thus can be considered as faultless. To save monitoring overhead, the debugging path is inactive by default, and will be activated when the module Fake-filter starts to report a victim-AS and a PoP.

Phase 3: identifying fault direction. Let l_{QoS} be the average packet loss that is monitored by QoS agents. Let l_{debug} be the average packet loss rate that is monitored by debugging agents. The fault debugger then identifies the fault direction according to the following formula.

$$Direction = \begin{cases} Inbound & \texttt{Cond}_1 \\ Outbound & \neg\texttt{Cond}_1 \wedge \texttt{Cond}_2 \\ Bidirectional & \neg\texttt{Cond}_1 \wedge \neg\texttt{Cond}_2 \end{cases}$$

$$\texttt{Cond}_1 = \left[l_{debug} \leqslant 3\%\right]$$
$$\texttt{Cond}_2 = \left[|l_{QoS} - l_{debug}| \leqslant 3\%\right]$$

The rationale behind the formula is as follows. 1) If it is an inbound fault, there should be no fault in outbound path and

debugging path, and therefore l_{debug} should be small, which corresponds to $\texttt{Cond}_1 = \left[l_{debug} \leqslant 3\%\right]$. Here, 3% equals to the constant δ (Section 4.3) that we use to filter minor faults. 2) If it is an outbound fault, both debugging agent and QoS agent should detect the fault, and therefore they must not meet $\texttt{Cond}_1$. Further, considering there is no fault in inbound path and debugging path, the difference between l_{debug} and l_{QoS} should also be small, which corresponds to $\texttt{Cond}_2 = \left[|l_{QoS} - l_{debug}| \leqslant 3\%\right]$. 3) If it is an bidirectional fault, as discussed in 1) and 2), it should not meet $\texttt{Cond}_1$ and $\texttt{Cond}_2$.

6 Detouring Victim Traffic

The detouring subsystem consists of a detouring controller and a BGP injector deployed on a per-PoP basis. We describe how the detouring controller and the BGP injector cooperate to detour traffic suffering from faults (so called victim traffic), circumventing outbound faults (§ 6.1) and inbound faults (§ 6.2), respectively. For bidirectional faults, we can split them into outbound faults and inbound faults, and then circumvent them separately. Therefore, we will not discuss how to circumvent bidirectional faults.

6.1 Circumventing Outbound Faults

Rationale: For every outbound fault associated with one victim-AS$_j$ and one reported PoP$_i$, the traffic from PoP$_i$ to victim-AS$_j$ is the victim traffic. To circumvent the outbound fault, considering its backup PoP$_{i+1}$ not reporting victim-AS$_j$, we decide to detour the victim traffic to egress at PoP$_{i+1}$. As the cloud network is fully under control, to achieve this, we can inject BGP routes to DRs in PoP$_i$. Because traffic detouring will inevitably degrade the latency when there is no fault, we need to detour the victim traffic back as soon as possible after the fault disappears. As AAsclepius deploys the QoS monitor on monitoring servers directly connected to PRs, the traffic detouring at DRs will not interfere with QoS monitoring. Therefore, the monitoring subsystem can continuously identify whether victim-AS$_j$ is suffering from faults after detouring, and thus we can detour the victim traffic back when we have high confidence that the fault has already disappeared.

Workflow: The workflow of detouring victim traffic proceeds as follows. First, to ensure safety, before detouring the victim traffic, the detouring controller checks whether the PRs in PoP$_{i+1}$ and the private backbone will exceed 80% load rate after this detouring. Here, AAsclepius can easily determine the load rate of the PRs and private backbone after traffic detouring because the traffic monitoring system built in Huawei Cloud (§ 2.1) shares its visibility to traffic volume between any PoP and any AS to AAsclepius. Second, if the checking result is safe, the detouring controller then collects all IP prefixes of victim-AS$_j$ from PRs in PoP$_{i+1}$ rather than PoP$_i$, so as to guarantee that the PRs in PoP$_{i+1}$ can route the detoured traffic to the destination IP address. Third, the detouring controller passes the collected IP prefixes of victim-AS$_j$ and the IP addresses of PRs in PoP$_{i+1}$ to the BGP injector in PoP$_i$. In PoP$_i$, the BGP injector maintains a BGP connection with each DR. Fourth, the BGP injector generates corresponding BGP routes for the received IP prefixes of victim-AS$_j$. For these routes, the `local_pref`[7] is set to a very large value (*e.g.*, 1000), and the `next_hop`[8] is set to the received IP addresses of PRs in in PoP$_{i+1}$. Fifth, in PoP$_i$, the BGP injector announces the generated BGP routes to all DRs. By setting `local_pref` to a large value, the generated routes can override the original routes, and then the victim traffic will be detoured to egress at PoP$_{i+1}$. Once victim-AS$_j$ has been identified as not suffering from faults continuously for **10 minutes** by the fault reporter in PoP$_i$ (§ 4.3), the detouring controller will then inform the BGP injector in PoP$_i$ to withdraw the corresponding routes, so as to detour the victim traffic back.

6.2 Circumventing Inbound Faults

Rationale: For every inbound fault associated with one victim-AS$_j$ and PoP$_i$, the traffic from victim-AS$_j$ to PoP$_i$ is the victim traffic. Similarly, we decide to detour the victim traffic to ingress at PoP$_{i+1}$. However, the PoP at which the victim traffic ingresses is directly selected by ISPs, not the CSP. To address this issue, we can change the BGP announcement of PoP$_i$, and leverage BGP to detour the victim traffic. In order for the QoS monitor to continuously monitor existing faults to provide guidance on when to detour the victim traffic back, the change of the BGP announcement should not involve the prefixes assigned to the QoS monitor.

Workflow: The workflow of detouring victim traffic proceeds as follows. First, the detouring controller performs the safety checking similar to that in circumventing outbound faults. Second, In PoP$_i$, the detouring controller informs the BGP injector to announce the dominating prefixes of PoP$_i$ to all PRs. Note that the BGP injector needs to prepend multiple duplicate AS'es to the `AS_path` of these prefixes. In this way, after BGP converges, the victim traffic will be routed to ingress at PoP$_{i+1}$. Note that the dominating prefixes of PoP$_i$ are also announced by the other PoPs with multiple duplicate AS'es prepended to their `AS_path` (§ 2.1). We should ensure that PoP$_{i+1}$ prepends relatively less duplicate AS'es to the `AS_path` of these prefixes.

Discussion: A major risk of changing the BGP announcement at PRs is that it detours not only the victim traffic, but also all the other inbound traffic of PoP$_i$ (we call them innocent traffic). The latency of innocent traffic will inevitably degrade. Currently, considering the inevitable side effects, we currently disable AAsclepius to execute automatic detouring for inbound faults. AAsclepius only notifies the network operators of inbound faults, and provides an API for traffic detouring.

[7] The BGP local preference attribute is the second BGP attribute that can be used to choose the exit path for an AS.

[8] The BGP next hop attribute is the next hop IP address that is used to reach a certain destination.

6.3 Discussion

We first discuss the benefits of identifying fault direction, which is the key novelty of AAsclepius. Then, we discuss the potential downsides in the additional path asymmetry introduced by AAsclepius.

Benefits of identifying fault direction: Identifying fault direction can help minimize impacted traffic during traffic detouring. Because the start point of outbound path (cloud → client) is under control, to circumvent outbound faults, we can accurately determine outbound traffic requiring traffic detouring. Because the start point of inbound path (client → cloud) is beyond control, to circumvent inbound faults, we must change BGP announcement, and all inbound traffic is impacted. Therefore, with fault direction, we can reroute traffic accordingly to minimize impacted traffic.

Negligible downsides in introducing path asymmetry: According to RFC 3349 [25], the additional path asymmetry introduced by AAsclepius during detouring victim traffic may degrade performance of TCP traffic. Nevertheless, path asymmetry is a common phenomenon in the Internet (87% path-tuples show path asymmetry [26]). AAsclepius has been running for years, without customers complaining about performance degradation. Further, some prior works (Meta Edge-Fabric [12]/ Microsoft CASCARA [27]) also introduce additional path asymmetry as they switch transit-providers for better performance or cost-effectiveness. Therefore, we conclude that the additional path asymmetry should have negligible impact on traffic performance.

7 Evaluation

We first present the deployment status of AAsclepius (§ 7.1). Then, we present the performance of monitoring subsystem, diagnosing subsystem, and detouring subsystem (§ 7.2-7.4). In addition, we select several typical faults as case studies to illustrate the workflow of AAsclepius (see Appendix A).

7.1 Deployment Status

We have fully deployed AAsclepius on a large scale in our country. In August 2020, we start to run AAsclepius for just some provinces. After a month of testing, we start to run AAsclepius for the whole country. So far, AAsclepius has been running stable for two years. AAsclepius has diagnosed thousands of faults and circumvented more than two hundred middle faults. In 2021 and 2022, AAsclepius protects Huawei Cloud from major accidents. Our SRE team identifies network faults that cause more than five VIP customers to experience more than 5% packet loss rate for 10 minutes as major accidents. Major accidents typically last several hours, involving tens of AS'es, with (i) construction-related optical cable cuts, (ii) router failures, and (iii) traffic congestion being main root causes. For example, in August 2022, an outbound fault (may lead to major accident) affecting three provinces began at 20:57, resulting in a packet loss rate of up to 40%. AAsclepius executes traffic detouring at 21:04 (within 8 minutes) to circumvent the middle fault, and the fault finally ended at 22:30.

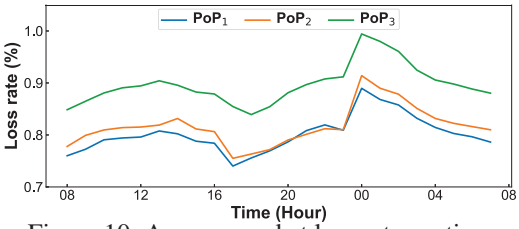


Figure 10: Average packet loss rate vs. time.

7.2 Performance of Monitoring Subsystem

We present the performance of the monitoring subsystem in three major PoPs in December 2021. First, we present the trend of packet loss rate and the distribution of victim-AS'es in different hours. Then, we present the distribution of fault duration of victim-AS'es. The following figures present data aggregated over 31 days in December.

Average packet loss rate vs. time (Figure 10): For the three major PoPs, we calculate the average packet loss rate of all IP*s in different hours. First, we find that the trend of the packet loss rate in each PoP is similar. This is possibly because the middle network in our country is adequately provisioned and well engineered, and thus the packet losses are mainly contributed by the client network. Because each IP* shares similar client paths in different PoPs, its packet loss rates in different PoPs are also similar, and thus the average packet loss rate in each PoP shares similar trends. Second, we find that the packet loss rate sharply increases to the peak at 0:00/24:00. It is possibly because ISPs in our country usually update routes and maintain network devices at this time, which is usually accompanied by network faults such as BGP misconfigurations, leading to the increase of average packet loss rate. Third, we find that the average packet loss rate in PoP_1 is slightly higher than that in PoP_2 and PoP_3. We suggest this is because that the middle network PoP_1 connected to usually has a relatively higher load.

Distribution of victim-AS'es vs. time (Figure 11): For each PoP, we count the distribution of victim-AS'es occurring in different hours. First, similar to the packet loss rate, we find that the distribution of the occurrence of victim-AS'es in each PoP is similar. Second, we also find that victim-AS'es are more likely to occur at 0:00/24:00. Third, we find that the distribution of the occurrence of victim-AS'es is positively correlated with the trend of packet loss rate. We suggest this is because a higher packet loss rate implies poorer network quality, which means more faults and thus more victim-AS'es.

Fault duration of victim-AS'es (Figure 12): Similar to Figure 4, we further present the distribution of fault duration of victim-AS'es in each PoP. We also find that the distribution is quite similar in each PoP. Considering that each PoP shares similar network quality, we can set unified parameters for each PoP to ease maintenance.

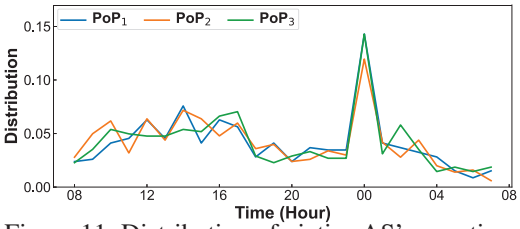


Figure 11: Distribution of victim-AS'es vs. time.

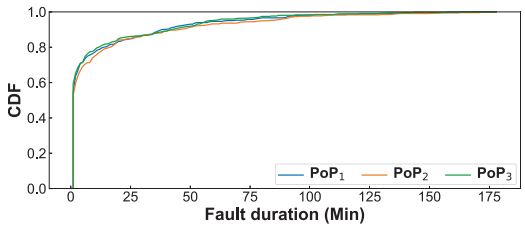


Figure 12: CDF of fault duration of victim-AS'es.

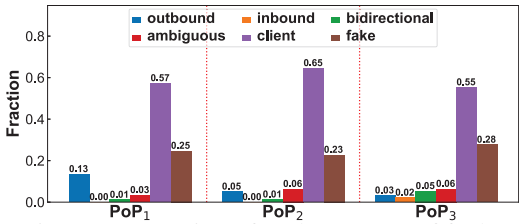


Figure 13: Fraction of each category of faults.

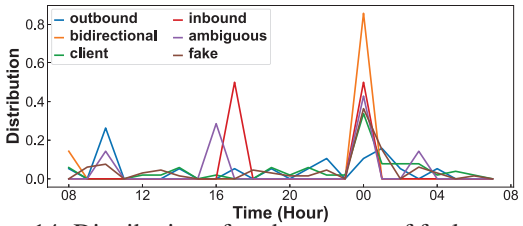


Figure 14: Distribution of each category of faults vs. time.

7.3 Performance of Diagnosing Subsystem

We present the performance of the diagnosing subsystem in three major PoPs in December 2021. First, we present the fraction of each category of faults. Then, we present the distribution of each category of faults in different hours. The following figures present data aggregated over 31 days in December.

Fraction of each category of faults (Figure 13): We count the fraction of each category of faults in each PoP. We find that the fractions in each PoP are similar: more than 50% of the faults are client faults, about 25% are fake faults, less than 15% are middle faults. It is possibly because the middle network in our country is well engineered and adequately provisioned, so that middle faults occur less frequently. We find that the fraction of outbound faults is far larger than that of inbound faults. The results show that outbound faults, inbound faults, and bidirectional faults account for 7%, 1%, and 2%, respectfully. We suspect this is because there is much more user download traffic than user upload traffic in the middle and client network, and thus the outbound paths are more likely to be congested.

Distribution of each category of faults vs. time (Figure 14): We count the distribution of each category of faults occurring in different hours. Similar to the distribution of victim-AS'es occurring in different hours (see Figure 11), we find that faults are more likely to occur at 0:00/24:00.

Potential misclassifications: Misclassifications are unavoidable. When there is a false positive (a non-middle fault is classified as a middle fault), AAsclepius may execute useless traffic detouring and increase the latency. When there is a false negative (a middle fault is classified as a non-middle fault), AAsclepius may not reduce the packet loss rate of victim traffic and receive complaints from customers. Because most network faults occur beyond our control, we can hardly obtain ground truth and misclassification rate. Nevertheless, AAsclepius can reduce packet loss rate in most traffic detouring (see Figure 16) and has protected cloud from major accidents for years, indicating an extremely low misclassification rate.

7.4 Performance of Detouring Subsystem

We present the performance of the detouring subsystem in three major PoPs in December 2021. We first present the response time of AAsclepius to middle faults. We then present the effect of traffic detouring on the packet loss rate and latency of victim traffic. The results show that the detouring subsystem is fast and effective. Note that we currently disable AAsclepius to execute automatic detouring for inbound faults, and thus all traffic detouring in this section is for outbound faults.

Evaluation criteria: In order to evaluate the detouring subsystem, we deploy *VM agents* in VMs in Huawei Cloud to perform *VM monitoring*. Similar to QoS agents, VM agents also send ICMP probes to each IP*. Because the probes sent from VMs are routed the same as the cloud traffic, we regard the performance of VM monitoring as the QoS, and use it to evaluate the effect of traffic detouring.

Response time to middle faults (Figure 15): We define the response time to a middle fault as the interval between the time its associated victim-AS transits from healthy-state to victim-state and the time the detouring subsystem reports the traffic detouring is executed. We find that the response time to all middle faults is within 8 minutes. Typically, the monitoring subsystem takes 3∼5 minutes to identify and report a victim-AS; the diagnosing subsystem takes 3∼4 minutes to identify its category and direction; the detouring subsystem takes less than 30 seconds to execute the traffic detouring.

Packet loss rate optimization (Figure 16): For each middle fault, we compare the average packet loss rate of its associated victim-AS in VM monitoring within 5 minutes before and after the corresponding traffic detouring is executed. We find that each traffic detouring reduces the packet loss rate by 7.0% on average. There are only less than 10% traffic detouring degrading the packet loss rate by less than 0.5%.

This is possibly because that the middle faults have already disappeared when the traffic detouring is executed.

Latency variation (Figure 17): For each middle fault, we compare the average latency of its associated victim-AS in VM monitoring within 5 minutes before and after the corresponding traffic detouring is executed. We find that each traffic detouring slightly degrades the latency by 1.9ms on average. This is reasonable because traffic detouring usually degrades the latency when no fault occurs, and the degradation has already been weakened by the middle faults.

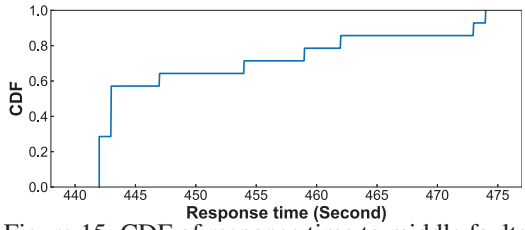


Figure 15: CDF of response time to middle faults.

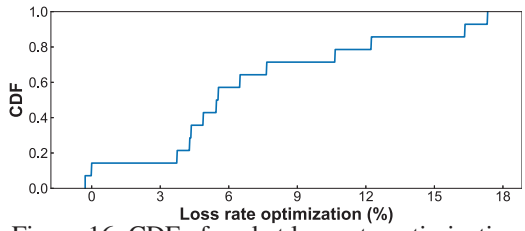


Figure 16: CDF of packet loss rate optimization.

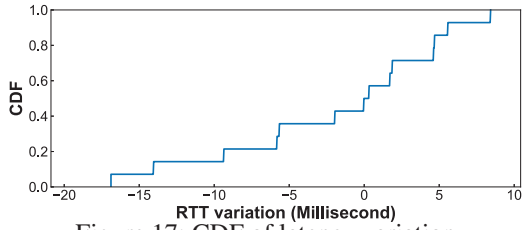


Figure 17: CDF of latency variation.

8 Related Work

Fault monitoring and diagnosis at Internet scale: Based on the measurement methods, existing fault monitoring and diagnosis solutions can be mainly classified into three categories: 1) active solutions which send probes to the Internet [14, 28–30]; 2) passive solutions which monitor ongoing connections [12, 13, 15, 17, 31, 32]; 3) hybrid solutions which combine active and passive solutions [1, 11, 16, 18, 19, 33]. Among these solutions, BlameIt [11] (hybrid), Entact [14] (active), Edge fabric [12] (passive), Espresso [13] (passive), and CPR [15] (passive) are most related to our application scenarios. BlameIt uses its passive measurement data to monitor faults and identify the fault category in the first phase, and further triggers impact-prioritized probes to localize the faulty AS for middle faults with the largest impacts in the second phase. However, BlameIt does not identify the fault direction, and is therefore distinguished from AAsclepius. The other solutions above do not diagnose faults but support traffic engineering, and we will cover them in the next paragraph.

Traffic engineering: There are substantial traffic engineering solutions, most of which are dedicated to optimizing CDN performance. A related kind of solutions select egress path and ingress point that a client should be directed to as a function of path performance [12–16, 34, 35]. Among them, Entact [14] measures path performance by sending probes to different IP /24 prefixes through alternate paths. Edge Fabric [12] and Espresso [13] passively measure path performance in different IP /24 prefixes by directing a small amount of flows to alternate paths and tracking their performance. Similar to AAsclepius, Espresso leverages Google's private backbone, B4 [36], and thus can route traffic to egress at distant PoPs. Through deployment in the kernel, CPR [15] even provides path failover at connection granularity. However, all the listed traffic engineering solutions optimize their traffic performance by selecting alternate outbound paths based on end-to-end measurement, and thus can only handle outbound faults. In contrast, AAsclepius detours traffic based on the category and the direction of the faults, and thus can handle both inbound and outbound faults. Therefore, these solutions are distinguished from AAsclepius. AAsclepius is also complementary to these solutions, as it can provide fine-grained fault category and fault direction information, which is useful for CDN performance optimization. Other solutions include IP anycast [37], co-located DNS and proxy servers [38], end-user mapping with EDNS [39], *etc.* [40].

Solutions in datacenters: Network faults in datacenters have been studied over decades, and researchers have provided various solutions [41–63]. These systems work excellently in data centers, but have not been extended to Internet scale.

9 Conclusion

Network fault is a widespread phenomenon in the Internet, which could harm the QoS of Huawei Cloud. Existing works do not identify fault direction. In this paper, we propose a fully automatic system, namely AAsclepius, to monitor and diagnose network faults, and detour victim traffic to circumvent middle faults. The key novelty of AAsclepius, PathDebugging, achieves identifying the directions of middle faults. AAsclepius has proven itself to be mature and reliable in two years of production deployment, and we consider extending AAsclepius to IPv6 network. Although the core methodology applied to IPv4 network can still be applied to IPv6 network, the main difficulty in the extension is how to efficiently find active IP addresses of high quality for QoS monitoring in the IPv6 address space which is much more larger than IPv4, and we are seeking for the solution.

Acknowledgment

We would like to thank the anonymous reviewers and shepherd Reto Achermann, for their help in improving this paper. This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, National Natural Science Foundation of China (NSFC) (No. U20A20179).

References

[1] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018.

[2] Rodérick Fanou, Francisco Valera, and Amogh Dhamdhere. Investigating the causes of congestion on the african ixp substrate. In *Proceedings of the 2017 Internet Measurement Conference*, pages 57–63, 2017.

[3] Matthew Luckie, Amogh Dhamdhere, David Clark, Bradley Huffaker, and KC Claffy. Challenges in inferring internet interdomain congestion. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 15–22, 2014.

[4] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 350–361, 2011.

[5] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM transactions on networking*, 16(4):749–762, 2008.

[6] Gianluca Iannaccone, Chen-nee Chuah, Richard Mortier, Supratik Bhattacharyya, and Christophe Diot. Analysis of link failures in an ip backbone. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 237–242, 2002.

[7] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[8] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 43–56, 2005.

[9] Daniel Turner, Kirill Levchenko, Alex C Snoeren, and Stefan Savage. California fault lines: understanding the causes and impact of network failures. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 315–326, 2010.

[10] Monia Ghobadi and Ratul Mahajan. Optical layer failures in a large backbone. In *Proceedings of the 2016 Internet Measurement Conference*, pages 461–467, 2016.

[11] Yuchen Jin, Sundararajan Renganathan, Ganesh Ananthanarayanan, Junchen Jiang, Venkata N Padmanabhan, Manuel Schroder, Matt Calder, and Arvind Krishnamurthy. Zooming in on wide-area latencies to a global cloud provider. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 104–116. 2019.

[12] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431, 2017.

[13] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445, 2017.

[14] Zheng Zhang, Ming Zhang, Albert G Greenberg, Y Charlie Hu, Ratul Mahajan, and Blaine Christian. Optimizing cost and performance in online service provider networks. In *NSDI*, pages 33–48, 2010.

[15] Raul Landa, Lorenzo Saino, Lennert Buytenhek, and João Taveira Araújo. Staying alive: Connection path reselection at the edge. In *NSDI*, pages 233–251, 2021.

[16] Matt Calder, Ryan Gao, Manuel Schröder, Ryan Stewart, Jitendra Padhye, Ratul Mahajan, Ganesh Ananthanarayanan, and Ethan Katz-Bassett. Odin: Microsoft's scalable fault-tolerant cdn measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 501–517, 2018.

[17] Brandon Schlinker, Italo Cunha, Yi-Ching Chiu, Srikanth Sundaresan, and Ethan Katz-Bassett. Internet performance from facebook's edge. In *Proceedings of the Internet Measurement Conference*, pages 179–194, 2019.

[18] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize cdn performance. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 190–201, 2009.

[19] Ming Zhang, Chi Zhang, Vivek S Pai, Larry L Peterson, and Randolph Y Wang. Planetseer: Internet path failure monitoring and characterization in wide-area services. In *OSDI*, volume 4, pages 12–12, 2004.

[20] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proc. ACM IMC*, 2004.

[21] Ajay Anil Mahimkar, Zihui Ge, Aman Shaikh, Jia Wang, Jennifer Yates, Yin Zhang, and Qi Zhao. Towards automated performance diagnosis in a large iptv network. *ACM SIGCOMM Computer Communication Review*, 39(4):231–242, 2009.

[22] Partha Kanuparthy and Constantine Dovrolis. Pythia: Diagnosing performance problems in wide area providers. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 371–382, 2014.

[23] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. *ACM SIGCOMM computer communication review*, 34(4):219–230, 2004.

[24] Junchen Jiang, Rajdeep Das, Ganesh Ananthanarayanan, Philip A Chou, Venkata Padmanabhan, Vyas Sekar, Esbjorn Dominique, Marcin Goliszewski, Dalibor Kukoleca, Renat Vafin, et al. Via: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 286–299, 2016.

[25] Hari Balakrishnan, V Padmanabhan, Godred Fairhurst, and Mahesh Sooriyabandara. Rfc3449: Tcp performance implications of network path asymmetry, 2002.

[26] Wouter De Vries, José Jair Santanna, Anna Sperotto, and Aiko Pras. How asymmetric is the internet? a study to support the use of traceroute. In *Intelligent Mechanisms for Network Configuration and Security: 9th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2015, Ghent, Belgium, June 22-25, 2015. Proceedings 9*, pages 113–125. Springer, 2015.

[27] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with cascara. In *NSDI*, pages 201–216, 2021.

[28] Lin Quan, John Heidemann, and Yuri Pradkin. Trinocular: Understanding internet reliability through adaptive probing. *ACM SIGCOMM Computer Communication Review*, 43(4):255–266, 2013.

[29] Ítalo Cunha, Pietro Marchetta, Matt Calder, Yi-Ching Chiu, Bruno VA Machado, Antonio Pescapè, Vasileios Giotsas, Harsha V Madhyastha, and Ethan Katz-Bassett. Sibyl: a practical internet route oracle. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 325–344, 2016.

[30] Harsha V Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: An information plane for distributed services. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 367–380, 2006.

[31] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 161–176, 2019.

[32] Venkata N Padmanabhan, Sriram Ramabhadran, and Jitendra Padhye. Netprofiler: Profiling wide-area networks using peer cooperation. In *International Workshop on Peer-to-Peer Systems*, pages 80–92. Springer, 2005.

[33] Vasileios Giotsas, Christoph Dietzel, Georgios Smaragdakis, Anja Feldmann, Arthur Berger, and Emile Aben. Detecting peering infrastructure outages in the wild. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 446–459, 2017.

[34] Hongqiang Harry Liu, Raajay Viswanathan, Matt Calder, Aditya Akella, Ratul Mahajan, Jitendra Padhye, and Ming Zhang. Efficiently delivering online services over integrated infrastructure. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 77–90, 2016.

[35] Vytautas Valancius, Bharath Ravi, Nick Feamster, and Alex C Snoeren. Quantifying the benefits of joint content and network routing. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, pages 243–254, 2013.

[36] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[37] Ted Hardie. Rfc3258: Distributing authoritative name servers via shared unicast addresses, 2002.

[38] Ashley Flavel, Pradeepkumar Mani, David Maltz, Nick Holt, Jie Liu, Yingying Chen, and Oleg Surmachev. Fastroute: A scalable load-aware anycast routing architecture for modern cdns. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 381–394, 2015.

[39] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM Computer Communication Review*, 45(4):167–181, 2015.

[40] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. Taiji: managing global user traffic for large-scale internet services at the edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 430–446, 2019.

[41] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, 2015.

[42] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. 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*, pages 662–680, 2020.

[43] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 595–612, 2017.

[44] Arjun Roy, Rajdeep Das, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Understanding the limits of passive realtime datacenter fault detection and localization. *IEEE/ACM Transactions on Networking*, 27(5):2001–2014, 2019.

[45] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based performance diagnosis for network functions. 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*, pages 390–403, 2020.

[46] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, 2016.

[47] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 481–495, 2016.

[48] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. 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*, pages 76–89, 2020.

[49] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016.

[50] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 ACM SIGCOMM Conference*. ACM, 2018.

[51] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *NSDI*, pages 991–1010, 2021.

[52] Kaicheng Yang, Yuhan Wu, Ruijie Miao, Tong Yang, Zirui Liu, Zicang Xu, Rui Qiu, Yikai Zhao, Hanglong Lv, Zhigang Ji, and Gaogang Xie. Chamelemon: Shifting measurement attention as network state changes. In *Proceedings of the 2023 ACM SIGCOMM Conference*. ACM, 2023.

[53] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. 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*, pages 404–421, 2020.

[54] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM CCR*, volume 45. ACM, 2015.

[55] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016.

[56] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo,

and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, 2018.

[57] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 440–453, 2016.

[58] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–8, 2018.

[59] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.

[60] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[61] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, 2019.

[62] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

[63] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, 2016.

A Real-world Case Studies

To better illustrate the workflow of AAsclepius, we present several typical faults as case studies.

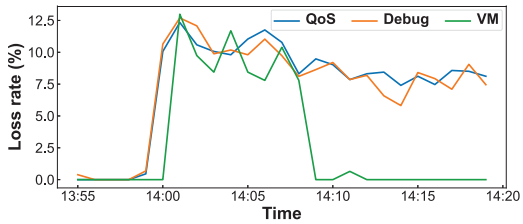


Figure 18: A typical outbound fault.

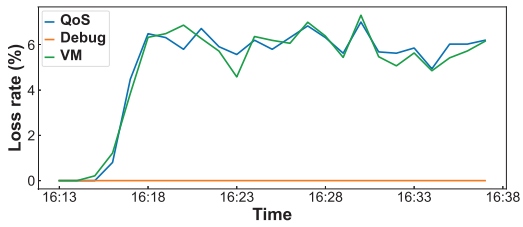


Figure 19: A typical inbound fault.

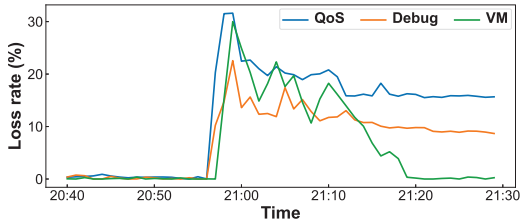


Figure 20: A typical bidirectional fault.

A typical outbound fault: Figure 18 plots the packet loss rates of QoS monitoring, debug monitoring[9], and VM monitoring during a typical outbound fault. The monitoring subsystem identifies a victim-AS at 14:00 at the first time, and reports the fault at 14:03. The diagnosing subsystem then identifies its fault category as a middle fault. The loss rate of debug monitoring keeps about 10%, which is similar to that of QoS monitoring. Therefore, the diagnosing subsystem identifies the middle fault as an outbound fault. The detouring subsystem detours the victim traffic at 14:08, and we can see the loss rate of VM monitoring suddenly decreases to almost 0%. In summary, the packet loss rate of VM monitoring decreases from up to 12.5% to almost 0% within 8 minutes. The outbound fault ends at 15:07, and the diagnosing subsystem detours the traffic back at 15:17, which is not plotted here.

A typical inbound fault: Figure 19 plots the packet loss rates of QoS monitoring, debug monitoring, and VM monitoring during a typical inbound fault. The monitoring subsystem identifies a victim-AS at 16:17 at the first time, and reports the fault at 16:20. The diagnosing subsystem then identifies its fault category as a middle fault. The packet loss rate of debug monitoring keeps less than 1%. Therefore, the diagnosing subsystem identifies the middle fault as an inbound fault. As we disable the automatic detouring for inbound faults, this fault is not circumvented, and the packet loss rate of VM monitoring keeps similar to QoS monitoring.

A typical bidirectional fault: Figure 20 plots the packet loss rates of QoS monitoring, debug monitoring, and VM monitoring during a typical bidirectional fault. This fault is a large-scale fault involving tens of victim-AS'es, and we just present one of them. The monitoring subsystem identifies the victim-AS at 20:57 at the first time, and reports the fault at 21:00. The diagnosing subsystem then identifies its fault category as a middle fault. The packet loss rate of the debug monitoring keeps around 15%, which is about 10% lower than that of the QoS monitoring. Therefore, the diagnosing subsystem identifies the middle fault as a bidirectional fault. The detouring subsystem detours the outbound traffic at 21:09, and the packet loss rate of VM monitoring decreases from about 20% to 10%. Due to the large scale of this fault, network operators manually detour the inbound traffic at 21:11, and the packet loss rate of VM monitoring decreases to less than 1% at 21:20. In summary, the packet loss rate of VM monitoring decreases from 30% to less than 1% within 20 minutes.

[9] We call the monitoring process performed by debugging agent in PathDebugging as debug monitoring for short.

Deploying User-space TCP at Cloud Scale with LUNA

Lingjun Zhu[*], Yifan Shen[*], Erci Xu[†], Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu

Alibaba Group

Abstract

The TCP remains the workhorse protocol for many modern large-scale data centers. However, the increasingly demanding performance expectations—led by advancements in both hardware (e.g., 100Gbps linkspeed network) and software (e.g., Intel DPDK support)—make the kernel-based TCP stack no longer a favorable option. Over the past decade, multiple parties have proposed various user-stack TCP stacks offering *things-as-usual* TCP support with significant performance improvement. Unfortunately, we find these proposals may not function well in the field, especially when subjected to large-scale deployments.

In this paper, we present LUNA, a user-space TCP stack widely deployed at Alibaba Cloud. We elaborate on the design tradeoffs, emphasizing three unique features in thread, memory, and traffic models. Further, we share our lessons and experiences learned from the field deployment. Extensive microbenchmark evaluations and performance statistics collected from the production systems indicate that LUNA outperforms kernel and other user-space solutions with up to 3.5× in throughput, and reduce up to 53% latency.

1 Introduction

At Alibaba Cloud, we follow a "compute-storage disaggregation" philosophy to enable the frontend computing servers (a.k.a Elastic Computing Service) and backend storage services (e.g., Elastic Block Storage, EBS) to evolve and scale separately. Initially, we adopted kernel TCP to connect computing servers to the storage servers for high compatibility and out-of-the-box usability.

However, advancement in hardware, including ultra-low latency (ULL) NVMe SSDs and high linkspeed networks (e.g., 100Gbps to 200Gbps), have significantly raised users' expectations for cloud storage systems. Kernel TCP is no longer a suitable option to deliver satisfactory performance, as it cannot fully leverage these benefits, and can lead to high tail latency and low single-core throughput. Moreover, kernel TCP can impose the well-known "data center tax" (e.g., consuming 70% of its CPU cycles in the kernel) [6, 20, 22].

Back in 2017, we started to notice such mismatches between the inefficient kernel TCP stack and the growing capabilities of new devices. We then began to look for an alternative solution to connect the frontend servers to backend storage systems. We explored the possibilities of replacing TCP with other protocols, such as Remote Direct Memory Access (RDMA), or leveraging hardware offloading (e.g., TCP Offload Engines). In fact, we have successfully deployed RDMA within our backend storage systems and achieved the expected performance gains [14]. Moreover, we have also designed a UDP-based protocol and leveraged Data Processing Units (DPUs) to accelerate one of our services, EBS [29].

Yet, interconnecting frontend and backend for all services is a different story. First, it requires inter-DC support to let storage services to be accessed from geo-distributed availability zones—not well supported by RDMA back then. Moreover, the interconnection network needs to provide compatibility and legacy support for various services, thereby prohibiting a complete overhaul with both the protocol altered and specialized hardware installed.

It is user-space TCP to the rescue. We noticed that a series of work, from both academia and industry, had demonstrated great performance potentials (e.g., saturating 40Gbps with IX [7]) by moving the TCP from the kernel to the user space [7, 9, 20, 21, 32]. More importantly, user-space TCP solutions provide a familiar programming model to the upper-level applications and offer legacy support by nature.

Unfortunately, we are unable to shoehorn existing user-space TCP solutions onto our production systems. First, these stacks normally use separate threads for application logic and the TCP processing (e.g., IX [7] and mTCP [20]), thereby incurring high communication overhead and impacting our Service Level Objectives (SLOs). Second, these solutions usually follow a copy-based memory model (e.g., mTCP and VPP [9]), aggravating memory bandwidth bottlenecks. Third, existing solutions require exclusively ownership of the NICs, thus preventing legacy support for kernel traffic.

In this paper, we present LUNA, a user-space TCP stack in Alibaba Cloud. We have successfully deployed LUNA in the field for more than 5 years and enable it to be the de-facto transport layer for all new servers in Alibaba Cloud since its release. Similar to previous practices (e.g., mTCP [20] and IX [7]), LUNA runs in a LibOS mode, operates in a shared-nothing architecture between threads and leverages DPDK user-space driver support.

[*] Equal contribution
[†] Corresponding author

Compared to previous practices, there are also three unique features in thread, memory and traffic models help LUNA successfully serve as an alternative to the kernel TCP. First, LUNA uses the run-to-completion (r2c) thread model. In each thread, LUNA packs both the application logic and TCP stack together, and process them with an event loop in each thread. Under this design, LUNA can significantly reducing the context switch overhead. Second, LUNA supports full data-path zero copy for both send and receive buffers based on a user-space slab subsystem. The zero-copy buffer effectively reduces the overhead from data movement. Third, LUNA can collaborate with kernel TCP stack to provide legacy support. We orchestrate the two types of traffic in the same NIC by utilizing Flow Bifurcation and SR-IOV to reserve certain ranges for user-space traffic.

We extensively evaluate LUNA against kernel TCP and two other user-space TCP implementations (mTCP [20] and VPP [9]) in a series of microbenchmarks. Results show that LUNA can outperform kernel and other user-space TCP stacks with up to 3.5× in throughput and reduce latency by up to 53%. Also, we compare performance between LUNA and kernel in the field across three representative scenarios. The field statistics show that LUNA could reduce latency by up to 50% and/or improve throughput by up to 50%.

The rest of the paper is organized as follows. We introduce the network architecture and the corresponding requirements at Alibaba Cloud (§2). We discuss the motivations behind LUNA in §3. We present the LUNA overview (§4) and three features in thread (§5), memory (§6) and traffic model (§7) designs. We further conduct series of evaluations on both microbenchmarks and field deployment (§8). We end this paper with several lessons we learned from deployment (§9) and a short conclusion(§10).

2 Background

2.1 Alibaba Cloud Storage Network Architecture

Alibaba Cloud offers various storage services, such as Elastic Block Storage (EBS), Object Storage Service (OSS), and Cloud Tablestore Service (OTS). In Figure 1, we illustrate the typical three layers of a service, including the interface, the function, and the persistence. In a nutshell, the interface layer comprises a set of servers to relay users' requests from the Internet (e.g., OSS and OTS) or the virtual stack within computing instances (e.g., vhost for EBS) to the function layer. The users' requests are further parsed and processed by the function layer (e.g., the BlockServers of EBS) before finally being sent to the persistence layer (i.e., Chunkstore Server, the storage engine of our distributed file system Pangu) for storage or retrieval. We use Pangu [25], an HDFS-like distributed file system developed by Alibaba Cloud, as the persistence layer.

Our cloud architecture adheres to a "compute-to-storage disaggregation" philosophy. This allows the computing

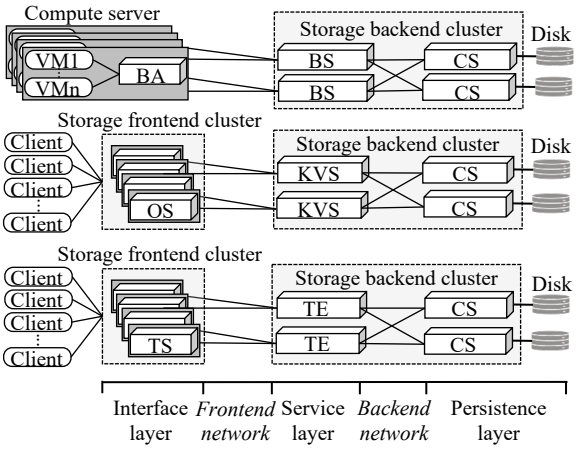


Figure 1: Alibaba Cloud storage network architecture. ***VM**: Virtual Machine; **BA**: Block storage interface Agent; **OS**: Object storage interface Server; **TS**: Tablestore interface Server; **BS**: Block storage function Server; **KVS**: OSS Key-Value Server; **TE**: Tablestore Engine; **CS**: Pangu distributed file system Chunkstore Server.*

servers—hosting the interface layer and virtual machine instances subscribed by the users—and backend storage systems (i.e., the function and persistence layers) to scale and evolve at different paces. In this case, we can divide our networks into two scopes, the frontend network and the backend network (see Figure 1). In the frontend network, we primarily use TCP. The backend network normally follows a two-layer Clos topology of Point of Delivery (PoD) and utilizes TCP or RDMA [14]. We focus on the frontend network in this paper.

2.2 Requirements for Our Networks

Following hardware evolution. With an ever-increasing user base, we are always in the process of expanding our fleet to accommodate more users and offering better performance. To achieve such goals, cloud vendors like us usually seek help from hardware advancement. For example, there are two aspects of recent development that fundamentally change the landscape of our data center networks. First, the network linkspeed has jumped from 10Gbps to 50Gbps and, more recently, 200Gbps. Moreover, high-throughput and low-latency storage devices become readily accessible. For example, off-the-shelf products, such as Intel P5800 [19] and Samsung Z-NAND SSD [36], can achieve up to 6GB/s throughput with around 10 microseconds latency. The combination of the two provides opportunities but also drastically raises the users' expectations of the cloud storage services.

Inter-DC access. A major difference between frontend and backend network is that the former needs to support computing servers accessing the storage servers from different clusters, data centers or even geographically far-apart availability zones. On the contrary, supporting the latter is relatively straightforward—normally a two-layer Clos of PoD.

Legacy support. Over the years, the sizes and types of our

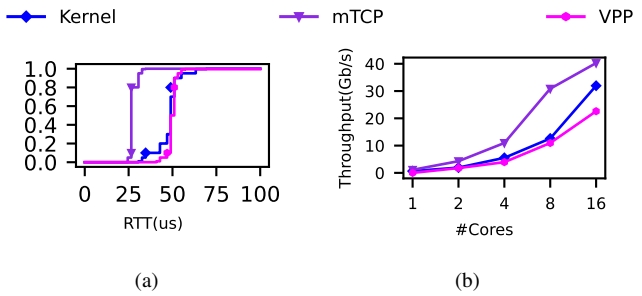


Figure 2: Different network stacks' performance in an offline RPC microbenchmark. a) RPC latency CDF with a single core; b) Maximal throughput and multi-core scalability.

storage services have been rapidly growing. Consequently, many outdated servers, while functioning, do not offer certain functionalities (e.g., DPDK-ready NICs), thereby may still require classic kernel traffic support.

3 Motivation & Related Work

In this section, we revisit the motivations behind LUNA. Although LUNA was initiated back in 2017, we believe the many observations and related work that inspired LUNA remain valid today and may even be strengthened over time.

3.1 Revisiting Kernel TCP

The kernel TCP was (in 2017) and may still remain to be a popular choice for large-scale clusters for its ease of usage and compatibility. However, it has become evident that the kernel TCP can no longer meet the performance demands of data centers (e.g., charging expensive data center tax due to kernel interruptions and memory copies) [6, 22, 33]. Here, we examines the kernel TCP performance via a microbenchmark.

We set up *knb* (Kuafu Network Benchmark)—an internal network stack benchmark that emulates the RPC services in the datacenters—to evaluate the kernel TCP latency and throughput. In this test, the client node sends a RPC request with 4KB messages to the server, measures the latency when receiving the RPC response with the same message size from the server, and then sends out the next request. (See § 8 for *knb* detailed usage.)

From Figure 2, we can see that the kernel TCP performance is far from our SLOs. Specifically, Figure 2(a) indicates the median basic RPC latency of kernel TCP has already reached 50μs. In stark contrast, our high-performance class EBS requires end-to-end response latency to be 100μs [3]. Figure 2(b) further shows that the kernel network stack on a single core could only provide a maximum of 600Mbps throughput. Moreover, Figure 2(b) also reveals that the performance issue of the kernel TCP cannot be resolved by allocating more cores. A possible reason is that the kernel overhead—such as inter-core competition—increases with the scaling of CPU cores.

3.2 Beyond Kernel TCP

Since the kernel TCP can be inefficient for the modern data center networks, it becomes urgent to explore possible alternatives. In general, there are three types of solutions: 1) developing new protocols, 2) moving TCP stack to user space, and 3) hardware offloading.

First, researchers have been proposing new transport layer protocols to replace TCP [4, 13, 16, 30]. pFabric [4] assigns priorities to packets based on the flow size, and tends to discard lower-priority packets in the switch when the buffer is full, thereby achieving both high throughput and low flow completion time. pHost [13], Homa [30] and NDP [16] leverage receiver-driven traffic control where the sender's sending rate is limited by the tokens sent from the receiver side. As each receiver has a global view of the incoming traffic, the receiver-driven protocols can achieve high bandwidth and low latency, and avoid the in-cast issue. For instance, Homa achieves less than 15μs for short messages on a 10 Gbps network running at 80% load. Moreover, these new protocols can eliminate the head-of-line blocking issue in TCP due to its byte-streaming nature.

Moreover, there are multiple work explore building a user-space TCP stack, such as mTCP [20], IX [7], ZygOS [35], TAS [23] and F-Stack [1]. These proposals usually leverage the user-space NIC drivers such as DPDK [2] to directly access packets from the NIC queues, and optimize the performance with techniques such as polling, batching [7, 20], cache planning [23], lock-free [7], and zero-copy buffers [7, 24]. The user-space network stacks minimize the overhead from the kernel, and demonstrate significant performance gain under the hardware advancement. For example, IX could achieve one-way latency of 5.7μs and fully utilize the 40Gbps bandwidth with 8 cores.

Third, there are several attempts aim to resolve the network performance issues with hardware assistance. For example, offloading TCP processing entirely [8, 37, 38] or partially [31] to specific devices can significantly improve packet processing performance and reduce the CPU overhead. RDMA [15] offloads traffic control and data movement to hardware, thus bypassing the CPU and achieving microsecond-level latency.

3.3 Our Choice

Among the available options, we chose to build a user-space TCP stack (i.e., LUNA) based on two aspects of reasons.

Inter-DC access and legacy support. Recall that, unlike the backend network, the frontend network needs to provide inter-DC access support (e.g., connecting computing servers from geo-distributed availability zones to storage servers). Therefore, we did not choose RDMA because it did not support inter-DC communication back then. Further, designing and deploying a new transport protocol (with hardware offloading) may also be rather challenging due to the required support for legacy software/hardware (e.g., sharing the NICs with kernel TCP traffic).

Table 1: Characteristics of existing user-space TCP

	kernel collaboration	zero-copy	high throughput	low latency
mTCP				
VPP			✓	
IX		✓	✓	✓
LUNA	✓	✓	✓	✓

Engineering effort. Note that enabling inter-DC access over RDMA or providing legacy support with the new protocol is still achievable but can be time-consuming. For example, the intra-region RDMA solution was not proposed until 2021 and a recent paper further discusses their strenuous effort on realizing wide-area RDMA accessing at Azure Cloud [5]. Back in 2017, we did have a rather tight timetable. Therefore, we chose user-space TCP stack solution to avoid designing/debugging the protocol and/or hacking support for legacy hardware. In fact, it only takes us 9 months to build LUNA from scratch to deployment. Later on, we have gradually added various optimizations and patches over the next five years (§ 9).

3.4 Why Not Just Use Existing Solutions?

Once we decided to use a user-space TCP stack for the frontend network, the next question is—"should we employ existing user-space TCP solutions or should we build our own?" Owning to the following reasons or concerns, we chose to develop a new user-space TCP stack, called LUNA.

High packet processing overhead. The microsecond-scale Service Level Objectives (SLOs) place significant pressures on packet processing speed within the network stack. Several existing user-space TCP solutions (e.g., mTCP and IX) delegate TCP protocol processing and application logic to separate threads for better portability. Meanwhile, others (e.g., VPP and TAS [23]) assign network and application processing to different cores for better scaling. Such partitioning could slow the processing speed due to context switch overhead or inter-core communication. For example, in Figure 2(a), we further profile the mTCP and VPP with the microbenchmark in §3.1. The results indicate that VPP suffers high latency, mainly introduced by the CPU cycles waste, and inter-core communication overheard.

Expensive memory copying. Data movement contributes a large proportion of datacenter tax. In a typical cloud storage service test on the 50Gbps network, memory copy can consume up to 12.5% CPU cycles, severely impacting the end-to-end latency (see §8). When the bandwidth grows to 100Gbps or more, the memory copy will take more than 40% CPU cycles, and further incur the memory bandwidth bottleneck problem. However, for user-space network stacks with a traditional IO path like mTCP [20] and VPP, there are two copy operations on the both receive and send paths (i.e., from user to TCP receive/send buffer and between TCP buffer to the packet). Figure 2(b) shows that, mTCP could not fully utilize the 50Gbps network bandwidth even with 16 cores. Our further analysis concludes that the memory copying does consequence with excessive overhead. One reason that most user-space TCP solutions do not support zero-copy buffer is that the traditional BSD-like socket interface would introduce inevitable memory copy between the application and TCP buffer for isolation.

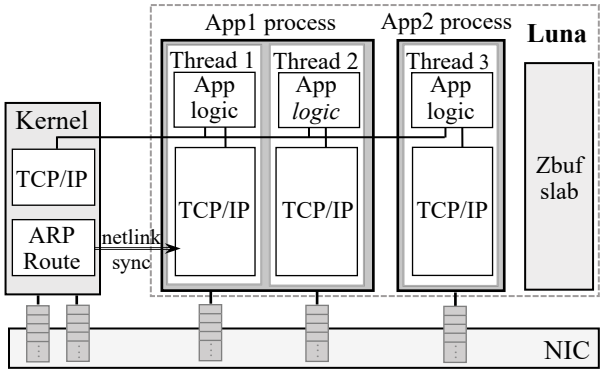


Figure 3: LUNA architecture

Supporting kernel traffic. Our storage services are deployed across multiple clusters consisting of several generations of machines. For many servers, their hardware are not capable of running user-space networks (e.g., lack of hugepages support). Moreover, many applications (such as monitoring agents) still rely on the kernel TCP. However, many user-space TCP stacks [1, 7, 20, 23] demand to exclusively own the entire NIC, thus could not collaborate with applications relies on kernel network stack on the same machine.

Implementation quality. Many existing user-space TCP works are research-oriented and thus can have various compatibility or performance issues. For instance, VPP is not well-compatible with Mellanox NICs when applying flow director filters. IX requires a particular Linux kernel version to run as it relying on the Dune kernel module, and also only provides drivers to the Intel NICs of outdated versions. Hacking into these problems will take an unexpected amount of engineering effort with rather limited community support. Hence, building a new user-space TCP from scratch can be actually more time-saving.

4 LUNA

4.1 Overview

LUNA is a high-performance user-space TCP/IP network stack that powers the frontend network in nearly all Alibaba Cloud Storage Services. Figure 3 shows that LUNA supports both kernel and user-space IO paths. LUNA leverages NIC multi-queues to separate user-space traffic from the kernel's. The u/ser-space IO path bypasses the kernel by leveraging the DPDK's user-space driver [2] to poll packets from the NIC

queues, processes them with a customized TCP/IP network stack, and interacts with applications mainly through an RPC library. The user-space IO path follows a run-to-completion mode, which runs iterations to complete both packet processing and application logic in a single thread on each core, and shares nothing between cores. LUNA separates kernel traffic and user-space traffic with NIC's hardware support. In this case, the kernel traffic for traditional applications remains unaffected. LUNA leaves the control plane (i.e., ARP table and route table management) to the kernel, and uses the *netlink* interface to access the route information.

4.2 Similar Design Choices

LUNA shares several similarities with existing work in the data path and the architecture. Here, we will discuss our rationales behind these design choices.

LibOS mode. LUNA operates in a LibOS mode, similar to the mTCP [20] and F-stack [1]. In this setup, the application and LUNA run in the same process and share the memory address space. Alternatively, one can run the network stack in the separate process, and communicate via shared memory. In this case, if there is only one network process in every server to serve different application processes, it is called the *Microkernel* mode (e.g., Google Snap [27]). Another solution is to have one network stack process for each application process, known as the *Sidecar* mode.

Although microkernel-based solutions at the industry level were not widely popular back in 2017, we later—after the launch of LUNA—have observed several instances adopting this approach. One notable example is Snap [27]—a user-space network stack deployed in Google's datacenter. One prerequisite for Snap is to closely follow the weekly release cycle of the network stack in Google Cloud. As a result, to avoid service interferences, the support transparent/live upgrade becomes indispensable. For microkernel-based solutions, enabling transparent/live upgrade is rather straightforward as the network stack is an independent process. Further, to achieve high efficiency, Snap also adopts Google's Pony Express transport protocol and one-side operation.

LUNA does not adopt the Microkernel or Sidecar mode due to performance concerns because both modes require frequent inter-process communication, incurring high runtime overhead. Note that LUNA still uses TCP for the transportation layer (see§3) and thus can not fundamentally modify the protocol like Pony Express or adopt one-side operation for high performance. We are aware that LibOS mode lacks live/transparent upgrade support as the network stacks with LibOS mode have to be compiled with the application code together. From our perspective, this disadvantage is acceptable as the TCP protocol is rather mature (i.e., not requiring frequent changes) and our storage services have periodical upgrading schedules. Hence, LUNA can just follow storage service upgrading roadmaps.

User-space NIC driver. Like most kernel-bypass network systems [1, 7, 10, 20, 21, 23, 32, 35], we build LUNA with DPDK [2] for its rich development kits and active community support. LUNA leverages DPDK's PMDs (Poll Mode Drivers) to directly access packets from the NIC queues. We also utilize DPDK's hugepage management, and data structures like hash map and *mbuf*. Further, as there are several generations of NICs in our cloud, the user-space driver provides a convenient way to communicate with them in a uniform interface.

Share-nothing architecture. To exploit the parallel processing capability of multi-core systems, like many previous designs [7, 20], LUNA runs the threads in a share-nothing mode. Each core processes its own traffic divided by the NIC's multi-queue technique, and finishes related application-layer processing on the same core. LUNA does not use a dispatcher mode (like TAS [23]) or load balancing (like task-stealing in Shenango [32]) due to cache efficiency and synchronization overhead (e.g., from lock and atomic operations) concerns. Note that there is already a service-level load balancing inside applications. The share-nothing architecture improves multi-core scalability, as the system can simply allocate more cores to applications to improve performance. Moreover, run-to-completion LibOS avoids the potential CPU cycles waste in dispateched mode as the user-space network stack has to keep polling the NIC queues.

TCP stack. As discussed in § 3, LUNA uses TCP as the transport layer protocol. LUNA implements TCP according to RFCs [28, 34], and supports congestion control, flow control, RTT estimation, and SACK. LUNA is compatible with other standard TCP stacks like Linux kernel network stack.

4.3 Unique Features

To better serve our systems in the field, LUNA also includes unique features from the following aspects:

Thread model. LUNA uses a run-to-completion thread model to run network and application-layer processing in the same thread. We use this design to improve the performance, and avoid risks in scheduling (e.g., thread-hang) with the characteristics of our storage service workloads (§ 5).

Memory model. LUNA supports a full data-path zero-copy buffer on receive and send end, aiming to minimize the data movement overhead. LUNA realizes its full-stack zero-copy with the aid of a user-space slab subsystem. This subsystem introduces little overhead and maintains the traditional programming model.

Traffic model. LUNA collaborates with kernel network stack, to offer the legacy applications with kernel TCP support, and to leverage kernel TCP for the control plane (e.g., ARP). LUNA uses the Flow Bifurcation and SR-IOV support to reserve a certain port range for user-space traffic, so that there is no interference with kernel traffic. The kernel network stack directly processes the control plane messages such as ARP requests and responses, and manages the control plane

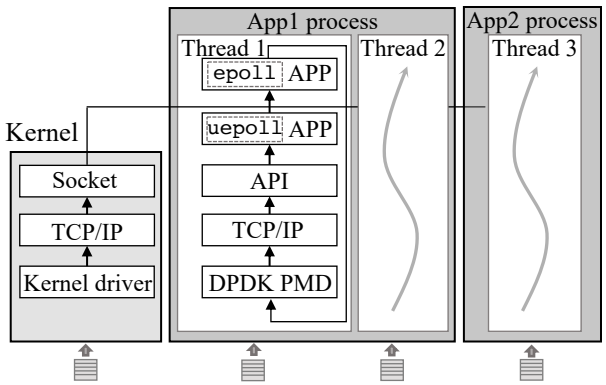


Figure 4: LUNA thread model

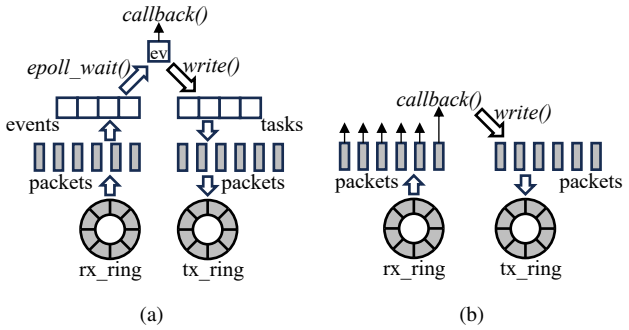


Figure 5: run-to-completion thread model in LUNA. a) batch-r2c mode; b) inline-r2c mode.

states. LUNA can obtain control plane information via *netlink* interface.

5 Thread Model

Figure 4 shows that LUNA uses a run-to-completion (r2c) thread model that encapsulates application logic and network stack processing in a single thread. During runtime, LUNA employs an event loop for each thread. Between threads, LUNA keeps a share-nothing isolation. Moreover, LUNA supports applications to use both kernel network stack and user-space IO path (§ 7), and the two types of traffic could be processed in the same thread.

5.1 Run-to-complete model

We assign varying numbers of cores to LUNA based on the service type. For instance, in a block server (i.e., the functional layer of EBS), LUNA can utilize 8 cores, while a computing server (from ECS) typically employs only 4 cores. Each core initiates one thread and shares nothing with the other. Moreover, each thread uses an event loop to manage data processing.

LUNA offers two distinct r2c modes——*inline-r2c* and *batch-r2c*——each tailored for different scenarios. In both r2c modes, LUNA starts the loop after receiving a fixed number of packets (called a batch) from the corresponding NIC Rx queue. Then, LUNA processes the packets based on the type of r2c mode.

For batch-r2c, Figure 5(a) shows that LUNA processes the received packets one at the time through the TCP/IP stack, and then adds a read event to the event queue for every packet with TCP payload. These read events would be immediately processed by application after the LUNA has processed all the received packets in this round. Then, the RPC framework invokes the callback functions registered to each event, generates the response messages, and sends the messages to the send buffer. After all the events are processed, LUNA adds the protocol headers for the messages in the send buffer, forwards them to the NIC, and starts the next round.

For inline-r2c, Figure 5(b) demonstrates that LUNA also processes the packets one by one. However, LUNA avoids adding event to the event queue, and instead immediately invokes the registered callback function, generates the response along with the protocol headers for the packets, and send them out. In short, inline-r2c will process every packet to completion.

Obviously, inline-r2c eliminates the overhead from event enqueue and dequeue, and improves the cache locality, thereby providing better performance. However, inline-r2c also requires a new programming model and forces the upper-layer application to use a zero-copy raw-packet-like read-/write interface. Moreover, inline-r2c is only available in LibOS model as the application-layer code has to co-locate with the network stack. In contrast, batch-r2c works in a more traditional epoll-like or libev-like programming model, and is compatible with a traditional BSD-Socket-like interface. In practice, we deploy inline-r2c on performance-oriented services like EBS, and use batch-r2c for services such as object store due to compatibility concerns.

The r2c design can significantly reduce the overhead and improve performance. First, there is no context switch between application and stack processing. Second, as the network stacks receive a fixed-sized batch of packets from NIC in each iteration, it allows the upper-layer application to handle them in a timely fashion (i.e., no need for buffering packets). Hence, CPU could get most data from the L1 and L2 cache directly, especially in inline-r2c. Moreover, as there are few buffered packets, the DDIO will not fill up the Last Level Cache (LLC), and further improves the cache hit rate [11].

5.2 Discussion

The risks of r2c model. R2c model could significantly improve performance. However, it is not favored by many user-space TCP stacks. A primary reason is that r2c model may be stuck at application level, causing severe tail latency and packet drop in NIC queues. LUNA adopts the r2c design because the logics in our storage services (i.e., applications) is rather simple and stable. Moreover, another safeguard is that our applications also adopt flow control and can avoid burst traffic by limiting the number of concurrent connections and in-flight requests. Hence, with relatively simple logic at

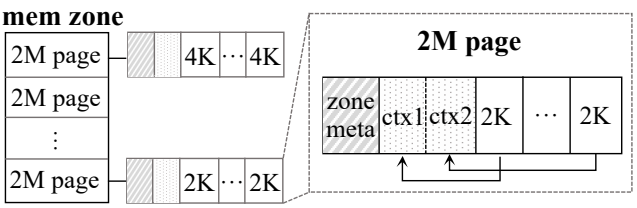


Figure 6: The structure of *Zbuf*.

the application level and flows throttled, it is unlikely for the requests to be stuck in the application level and cause packet drop. Note that it is still possible for a thread in LUNA to get stuck by various exceptions (e.g., application bugs, burst incast load, etc.), resulting in dropping packets and TCP/RPC timeouts.

Unsuitable scenarios for r2c model. The LUNA's r2c design is not always suitable for all scenarios. In OSS service, we discover that dedicating multiple cores to polling for run-to-completion is not reliable because there are other services on the same machine that should be guaranteed with a large number of cores. Therefore, LUNA dedicates only one core for NIC IO and protocol processing, and places application logic to other threads on the other cores. The application thread will block the event-poll when there is no more events to avoid wasting CPU cycles on unnecessary polling.

6 Memory Model

LUNA achieves full-stack zero-copy to mitigate the overhead associated with frequent data movement. A straightforward approach to realize end-to-end (i.e., from NIC to TCP, and application) zero-copy is to only transfer the memory addresses of the read/write buffers. This is challenging for user-space TCP due to three factors. First, the lifecycle (and status) of a buffer is different from the application to the network. The application typically frees or reuses the buffer after dispatching it to the network stack, whereas the network stack must retain the buffer until it receives "acks". Second, the NIC requires the physical memory address while applications use the virtual address. Third, the traditional BSD-socket APIs and socket-oriented programming models are designed with copy semantics for the isolation between the user space and kernel space. To overcome these challenges, in LUNA, we build a user-space slab system, called *Zbuf*, to provide cross-layer memory lifecycle management and address translation.

6.1 *Zbuf*

User-space slab subsystem. *Zbuf* works as a user-space slab subsystem that pre-allocates memory chunks for users. Figure 6 shows the structure of *Zbuf*. We can see that *Zbuf* reserves several hugepages allocated from the DPDK's memory address space and divides them into multiple 2MB *memory zone*s. The header of each *memory zone* records the meta information such as the physical address. *memory zone* is further split into *obj*s, which could be directly allocated by the

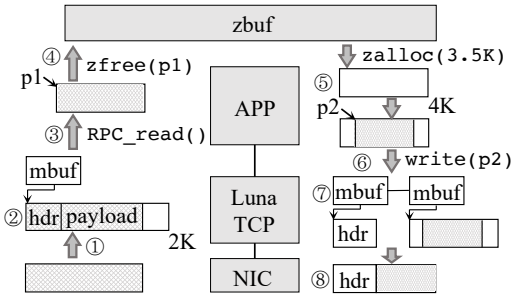


Figure 7: The LUNA receive and send path with zero-copy.

users (i.e., applications). The metadata of *obj*s (denoted as ctx in Figure 6) is co-located with *obj*s in the same *memory zone*, right after zone metadata. All *obj*s within the same *memory zone* share the same size but the sizes can be different from one *memory zone* to another (e.g., 2KB vs. 4KB in Figure 6).

***Obj* lifecycle management.** *Zbuf* uses a reference counter to manage the lifecycle of each *obj*. The counter is set to 1 after initialization. Afterward, the counter increases by 1 whenever the corresponding *obj* is replicated and decreases by 1 each time the *obj* is freed. The *obj* will be put back to the free-list of the *memory zone* once the count reaches 0.

Metadata and physical address translation. Translating the virtual address to the physical address and parsing the metadata are straightforward in *Zbuf*. For example, consider a user allocating a 4KB *obj*. The user then generates a 2KB string *str* within the *obj*, and the virtual address of *str* is *addr*. When the user sends *str* to the network stack, it would increase the reference count of the corresponding *obj*. *Zbuf* first compares *addr* with the address range of contiguous *memory zone*. By getting the offset of *addr* to the start of *memory zone* area and dividing the offset with 2MB, *Zbuf* could get the index of the *memory zone* which *obj* belongs to. As the *memory zone* metadata records the start address of contiguous *obj* and the size of each *obj*, the user could directly get the index of the *obj* containing the *str*, and get the *obj* meta data from the *obj* meta array. Therefore, the users could directly make replicas or free the objects inside the *obj*s, but do not need to manage the *obj*. Since the metadata of *memory zone* records the physical address of itself, the physical address of *str* could be calculated by adding the offset of *str* to the *memory zone*. When the *str* is going to be sent to the NIC, LUNA can calculate the physical address following the same procedures above.

6.2 Full-stack Zero-copy

With the support of *Zbuf*, LUNA provides full-stack zero-copy on data receiving and sending. Moreover, upper-layer applications could still use the traditional programming model except for a few minor changes. Now, we use Figure 7 to illustrate the procedures on both ends.

On the receiving end, LUNA registers *obj* addresses to NIC receive queue, so that the NIC will deliver the received

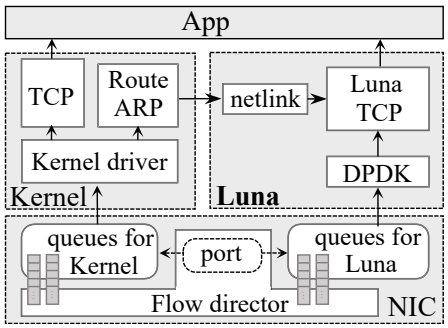


Figure 8: LUNA traffic model

packet data to the *obj* through DMA (①). The network stack processes the packet with network protocols (②) and delivers the pointer to the readable payload (marked as $p1$ in figure 7) to the upper-layer application with an RPC interface (③). The application could directly free the payload after finishing the message processing (④). *Zbuf* will locate the related *obj* and free it by decreasing the reference count.

On the send data path, application first allocates a writable buffer from the *Zbuf*, or reuses the memory space of the received data, and then directly writes data to the buffer (⑤). The content (marked as $p2$ in figure 7) is directly sent to LUNA's TCP send queue, and the send API automatically increases the reference count of the related *obj* (⑥), to avoid the *obj* to be freed by the application after sending. LUNA allocates a new segment to generate the packet header (⑦), and sends out the header and content as a whole packet together through the DPDK interface (⑧). Moreover, if the application wants to multicast the data (typical in cloud storage service), it just needs to send out the same data segment multiple times, and *Zbuf* will accordingly increase the reference count of the *obj*. And the *obj* stored in the LUNA's send buffer will be freed after receiving the acknowledge packet.

6.3 Discussion

Alternative solutions. There are also other design choices to achieve end-to-end zero copy. One is to pass the pointers of the read/write buffers between the application and network stack. However, adopting this practice requires the application to fundamentally alter the programming model and manage the buffer at application layer. For cloud services, it can be rather difficult as there are multiple applications developed by various developer teams. Another approach is to leverage *mmap* to avoid copying large size messages [24], which could maintain the programming model and standard BSD-socket API format. However, the *mmap* call can introduce considerable overhead of additional system calls [24].

7 Traffic Model

7.1 Traffic Split

LUNA uses Flow Bifurcation mechanism [18] (supported by NIC's flow-director) and SR-IOV functionalities to separate kernel network traffic from the user-space network traffic, thereby offering legacy support. LUNA establishes the hardware filtering by setting the mask to the destination port of TCP packets on each machine. LUNA routes the incoming TCP packets with certain destination ports to the specific virtual functions, which are then processed in user space. The TCP packets that do not align with the port filters and the not-TCP packets would still be accepted and processed by the kernel network stack.

LUNA uses different flow-director filter rules for clients and servers. On the client side, LUNA reserves the port from 61440 to 65535 for the user-space traffic, and allocates contiguous sub-ranges of the port number to different LUNA applications. The sub-range within an application are further divided into ranges for different Rx queues which are processed by the corresponding LUNA threads.

For instance, LUNA could reserve TCP port numbers between 61440 to 63487 to APP_A, and write the flow-direct filter rules to direct dest port between 61440 to 62463 to $Thread_1$ in APP_A. Then LUNA can direct TCP ports between 62464 to 63487 to $Thread_2$ in APP_A. On the server side, LUNA first uses flow-director rules to direct TCP packets with the same destination port as the application-listening port to the corresponding application. Moreover, the RPC layer over LUNA will establish full-mash connection for all thread peers between each client node and server node. Hence, LUNA on the server side can simply hash all connections to different server threads according to the source TCP port for load-balance scheduling. For example, the TCP destination port of 1234 is directed to APP_A, and uses the lowest 2 bits of the TCP source port to hash the packets to the 4 threads of APP_A. Each client thread initializes connections with typical host ports to establish a connection with every server thread, and selects a connection for each RPC request for load balancing.

Further, although LUNA is compatible with standard TCP stacks in the design and implementation, LUNA avoids directly communicating with the kernel network stack. In other words, LUNA only supports kernel-to-kernel and LUNA-to-LUNA traffic, and does not permit kernel-to-LUNA connection (or reverse). The reason for this choice is that LUNA implements a tailored TCP for the datacenter environment to optimize the performance (§ 9), and there are different versions of LUNA running in different datacenter applications. If we use LUNA to directly communicate with kernel network stack, we have to verify and evaluate the communication among all versions of kernels and the LUNA. Therefore, this introduces extra verification costs every time updating the kernel or modifying the kernel network configuration.

7.2 Thread Model Support

When the applications need to communicate with different clusters through both kernel and user-space IO paths, LUNA will process them in the same thread, so that the applications don't need to manage the requests separately. In every

iteration, after finishing the user-space run-to-completion processing, LUNA will get a batch of events from kernel's *epoll* framework by calling *epoll_wait()*, and handle the events by calling the callback function registered by the application. LUNA calls *epoll_wait()* with non-blocking, and limits the batch size of kernel events to prevent user-space IO path from starving.

7.3 Control Plane

As LUNA only processes the TCP packets, the control plane packets (e.g. ARP and ICMP requests) are sent to the kernel network stack, and the kernel also manages the control plane states (e.g. ARP table and route table).

LUNA gets the control plane information with *netlink*, a rather infrequent behavior. LUNA initializes a netlink socket which dumps the ARP table and route table in the kernel, and wait event through linux *epoll*. When there are any variations in these two tables, the kernel will send messages to the netlink socket and raise epoll events. Then this would invoke LUNA to receive the update message and update the control plane information managed in user space.

Delegating the control plane to the kernel brings two benefits. First, LUNA could focus on the transport layer and leverage the well-developed kernel control plane implementation. This also enables the LUNA to evolve independently without worrying about keeping up with changes in the control plane. Second, this allows LUNA to fast recover from the system failures, as the just-restarted LUNA could simply regain the control plane information stored in the kernel.

7.4 Discussion

Alternative solutions. There are several approaches to collocating user-space traffic within kernel network stack. The first one is to separate different NICs (or different ports of the same NIC) for user-space traffic and kernel network traffic. Unfortunately, this can severely waste bandwidth. Another approach is to receive all the packets using the user-space TCP and then re-dispatch certain filtered packets back to the kernel via KNI (Kernel NIC Interface). This solution can also impact the performance and crash the whole network service when the failure occurs in user-space network stack.

Complex filtering rules. LUNA splits the traffic according to the TCP ports to collaborate with the legacy applications and employs share-nothing design between cores for high performance. However, the traffic splitting is limited by the NIC hardware capabilities. The commodity NICs (e.g., Intel and Mellanox NIC cards) provide limited flow director support, i.e., setting masks to certain fields of the packet headers (e.g., IP address and TCP/UDP port number).

In LUNA design, one application keeps the same listening port for all server threads to cater the existing programming models. And the LUNA RPC framework establishes full-mash connection channels between every thread of each peer node for load balancing. Therefore, LUNA has to write multiple flow director filter rules to the NIC to spray the traffic to different Rx queues, and the number of rules increases dramatically when the number of LUNA threads is not a power of 2. For instance, when a machine running 12 LUNA threads communicate with a peer node running 6 LUNA threads, this will lead to 160 traffic filter rules on both nodes, imposing significant overhead onto the packet receiving, and resulting in packet drops.

One simple approach to solve this problem is to assign different listening ports for each LUNA thread. However, this also requires modification of the application logic. A more practical way is to reserve range of the port numbers with flow-director, and perform build-in hash-based RSS to establish the connections to different LUNA queues. Yet, this feature is not supported by the commodity NICs when the LUNA was developed. Once the flow bifurcation rule is deployed, we have to provide legacy support for previous versions for compatibility. As a result, LUNA still requires complex filtering rules at the moment.

8 Evaluation

8.1 Microbenchmark Evaluation

8.1.1 Experiment Setup

We first evaluate with microbenchmarks in the emulated client-server environment to compare LUNA against other candidates including kernel TCP, mTCP and VPP. Both client and server machine are equipped with an Intel(R) Xeon(R) CPU E5-2682 v4 @ 2.50GHz CPU with 128GB DRAM each. We connect the client with server through a Mellanox ConnectX-4 Lx NIC with 2× 25Gbps network port, and utilize both the port for a total 50Gbps bandwidth.

For the kernel network stack, we use Linux 4.19, and bind all traffic to the certain CPU cores to optimize the performance. We download the latest versions of mTCP and VPP, and modify them to fit our environment. For VPP, we use half of the cores for the VPP threads processing network packets, and use the rest cores for application logics. As for the common hardware offloading, we enable both *tso* and *lro* for Linux kernel, and enable *tso* for LUNA. For mTCP, *tso* is not supported, and *lro* is not supported by default. For VPP, we failed to enable *lro* and *tso* on our cx4 NIC with default driver after a series of attempts. And the MTU is 1,500 bytes for all the systems.

We use *knb*, a datacenter network microbenchmark, to evaluate the performance of LUNA and the rest. *knb* emulates the RPC workloads in the datacenter, and evaluates the network stack performance at both client and server side. *knb* runs a configured number of threads at client and server, and builds long-lived TCP connection between every client and server thread, similar to most data center RPC frameworks. Then, the *knb* client will send requests with configurable message size to the server. Afterwards, the server send back the response with the same message size on each request, and the

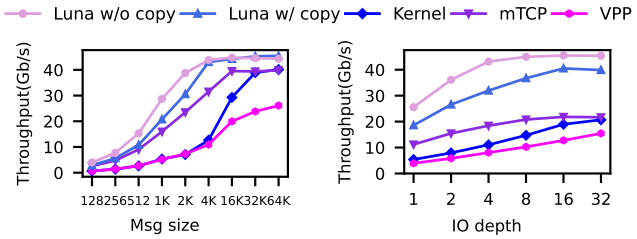


Figure 9: Different message sizes (*iodepth* =1, #core=8)

Figure 10: Different *iodepth* (msg_size=4KB, #core=4)

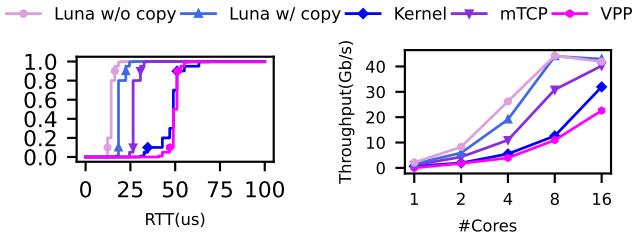


Figure 11: Response latency CDF (msg_size=4KB, *iodepth* =1)

Figure 12: Multi-core scalability (msg_size=4KB, *iodepth* =1)

client will send another request when it receive a response. The number of in-flight requests on each connection could be controlled with the variable *iodepth*.

8.1.2 Experiment Results

Throughput We first evaluate the throughput under different message sizes. Here, we set LUNA with both the copy mode (backed by batch-r2c) and the zero-copy mode (backed by inline-r2c). In this experiment, we allocate 8 cores for each candidate, and limit iodepth to 1 (i.e., only one request on each connection, and do not send next request until receiving the response). The Figure 9 shows the result. LUNA with zero-copy can fully utilize the bandwidth when the message size grows to 4K, a typical size in our storage services. The throughput of LUNA is 3.5× of kernel, and outperform mTCP and VPP by 50%. The mTCP and VPP do not fully utilize the bandwidth in these experiments. For mTCP, its performance is limited by the heavy overhead from copy and context switch. As for VPP, when the packet size is small, it shows a similar performance with kernel network stack, mainly as a result of wasted CPU cycles on idle polling. When packet size grows larger, VPP shows a even worse performance than Linux kernel (possibly due to *lro* and *tso* are not enabled in VPP).

We also evaluate the throughput under different setups of iodepth as RPCs in datacenters are always concurrent on the same connection. In this experiment, we dedicate 4 cores for each candidate (commonly seen in servers of the interface layer), and set the message size to be 4KB. Figure 10 shows that the throughput grows with the increasing of iodepth. Moreover, the zero-copy version of LUNA can saturate the bandwidth with an iodepth of 16. LUNA provides 2× throughput than mTCP and kernel network stack.

Latency We then evaluate the latency of the network stacks, and show the results in Figure 11. In this experiment, we allocate one core for VPP network worker thread and one core for *knb* application thread. Then, we use a single core when test other network stacks. We set the iodepth as 1 (i.e., no backlog blocking latency) and set the message size to be 4KB. The result shows that, LUNA with zero-copy reduces the 99^{th} percentile latency by 25% than LUNA with copy, and reduce 70% latency than the kernel network stack.

Multi-core scalability. In Figure 12, we evaluate the multi-core scalability of the tested network stacks. In this experiment, we limit the iodepth to be 1, and set the message size to be 4KB. We can see that LUNA shows a near linear multi-core scalability, and full-utilize the bandwidth with 8 cores with zero-copy. mTCP shows relatively good multi-core scalability as it also uses the share-nothing architecture. Yet, it still could not fully saturate the bandwidth due to the performance limitation. The kernel network stack and VPP show a similar multi-core scalability because there are extra inter-core communication overhead and contention over locks between different cores.

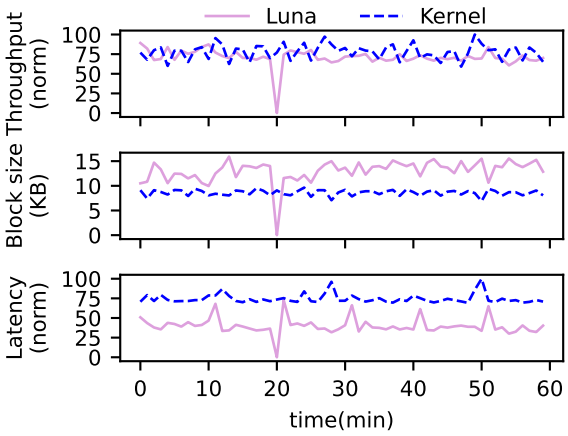


Figure 13: Performance of distributed-storage service for a public cloud OLTP service, collecting tracing data within 1 hour.

8.2 LUNA Performance in the Field

In this section, we will introduce the performance of LUNA with the datacenter storage services in the wild, and make comparison with the traditional kernel network stack. Since LUNA has been deployed in Alibaba Cloud storage service for more than 5 years, most servers that require high-performance are running on LUNA instead of kernel network stack. Therefor, we only show the performance comparisons in the services that still have legacy nodes running the kernel network stack.

EBS for OLTP. OLTP (On-Line Transaction Processing) ser-

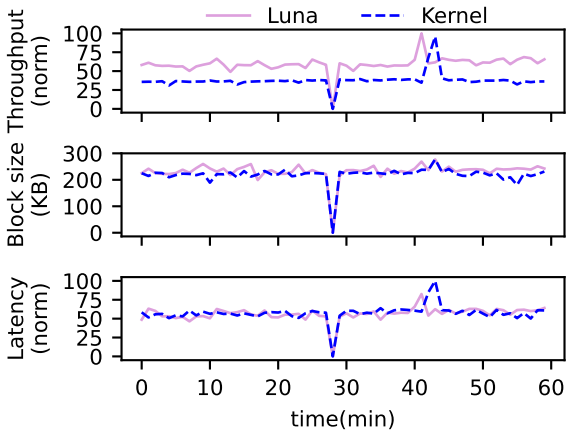


Figure 14: Performance data of distributed-storage service for a public cloud video transcoding service, collecting tracing data within 1 hour.

vice generate typical latency-sensitive workloads for the cloud storage system. Under such workloads, the IO throughput of the storage are usually not very high, but have rather demanding requirements on latency. Here, we show the LUNA deployment in a storage cluster serving a public cloud OLTP service. In these clusters, there are several generations of machines. The LUNA is only deployed to the server nodes that meet the hardware requirements, and still use the kernel network stack in the other legacy server nodes. Note that this cluster uses different network stack IO path only in the frontend network communicating with the block servers, but uses the same version of the backend network and the chunk servers.

In figure 13, we demonstrate the performance of the distributed-storage IO which is captured by the hypervisor-level monitoring during 1 hour. We can see that LUNA outperforms the kernel network stack with both 50% lower end-to-end average latency with similar end-to-end throughput. This gain mainly benefits from the thread model design and zero-copy support therefor reduces the processing and queueing delay. Note that the workload over LUNA has larger IO block sizes as they are from different customers, thereby higher pressure on the delay.

EBS for video transcoding. Here, we show LUNA performance on another EBS cluster which serves a public cloud video transcoding service. In this scenario, the datacenter application requires both high throughput and low latency. This cluster also uses both LUNA and kernel network stacks to communicate with blockserver agents (BAs) in the frontend network. Similarly, the backend network shares the same backend network and chunk server (CS) architecture. all the server nodes share the same workloads. We also collect and compare the user-layer performance data at the hypervisor. Figure 14 indicates that, when the service node with LUNA and kernel network stack have the similar user-level IO latency, LUNA provide about 50% higher IO throughput than the kernel network stack.

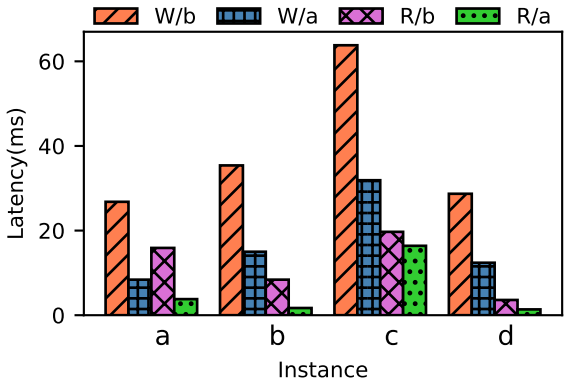


Figure 15: OTS response latency comparison before and after an architecture upgrade, collected from several typical instances.

TABLE STORE. LUNA is deployed in TABLE STORE, a NoSQL database service of Alibaba Cloud. TABLE STORE provides storage and real-time access to massive structured data. Here, we collect and compare the performance data between two generations of the TABLE STORE architecture, named V1.0 and V2.0. The workload remains unchanged after the architecture upgrade. The architecture evolution mainly includes using LUNA user-space network to replace the Linux kernel at both the frontend network and the backend network, and using the user-space file system to replace the Linux ext4. In our offline estimation, LUNA contributes 30%-50% to the total performance gain. Figure 15 shows that, TABLE STORE service instances upgraded to V2.0 reduce the end-to-end latency by 50% to 68%.

9 Lessons From Deployment

Portability. We believe there are four levels of portability.

- Kernel-based applications do not need any code changes to use the new stack (e.g., LOS [17]).
- The application needs to replace the APIs while keeping the same API formats and semantics.
- The API formats are different but the programming model and the API logic stay the same (e.g., change malloc/free to create/destroy).
- Redesign the entire programming model and the API logic.

Our lesson is that, to achieve extremely high performance, refactoring legacy application (i.e., programed with the kernel network stack) is inevitable. However, as a fundamental component of datacenter software codebase, the network stack is often widely used and serves various kinds of applications. Hence, changing the programming model is unacceptable. In this case, we would recommend a user-space network stack to obtain portability between 2) and 3).

In Alibaba Cloud storage, the applications attach to the network through an RPC framework which is also supported

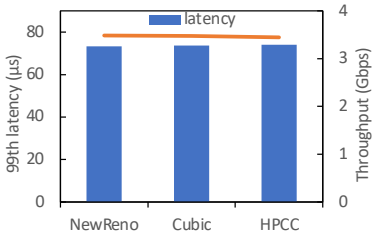


Figure 16: Performance microbenchmark of different congestion control algorithms tested in Alibaba Cloud cloud storage network.

by the network group, and programed in a *libevent-like* model, so adapting to the batch-r2c and inline-r2c does not require any change to the programming model. However, in order to achieve zero-copy at both receive and send side, LUNA provides io-vector-oriented receive/send APIs which directly describe the addresses and lengths of the readable/writable buffers of each RPC call. Therefore, the applications with zero-copy have to change the receive/send API formats. Nonetheless, the data buffers can be allocated or freed as if they are allocated from the heap through the *Zbuf* APIs with the same formats.

TCP tailoring. With a relatively simple environment as the datacenter network, we can tailor the implementation of TCP to achieve better performance. For example, LUNA implements a straightforward yet high-performance TCP fast path for the packets arriving in order. Note that out-of-order packets are not common in the datacenter. In our practice, we found that simply using NewReno [12] could deliver satisfying congestion controlling in many services and does not rely on any novel hardware features (Figure 16). Additionally, we also use HPCC [26] for the communication between VMs in the computing cluster to improve tail latency.

Fast recovery from the failures. For high availability, the network stack should fast recover from the failures such as switch flips and black holes. In our services, applications use LUNA via an RPC framework, which detects connection failure and tries to reconnect the peer node through different network links, e.g., another NIC port, or ToR switch. Further, LUNA also improves the failure recovery procedures based on the characteristics of the datacenter environment. For example, the network distance of every two nodes in our cluster is no more than 4 switch hops. As a result, LUNA could set a tighter timeout threshold, e.g. 4 milliseconds. Currently, the longest recover time for LUNA is guaranteed to be less than 2 seconds.

Alibaba Cloud storage network evolution. The evolution of LUNA is driven by service. Back in 2017, Alibaba Cloud planned to launch a high-performance ESSD service—the first elastic block storage service achieving both 1M IOPS and 100μs latency—around early 2018. With the release date within a year ahead, we therefore chose user-space TCP stack solution to avoid designing/debugging the protocol and hacking support for legacy hardware. It only takes us 9 months to build LUNA from scratch to deployment.

In the initial release, LUNA still uses socket-like APIs and requires data copy from the application to the network stack on the send path. Later, to support the bare-metal servers, LUNA needs to run on a Data Processing Unit (DPU) which has rather limited resources. Therefore, we designed a new RPC framework which removes the RPC serialization stage, supports the inline-r2c thread model (§ 5) and the zero-copy IO on both ends (§ 6). Note that this also requires the EBS to change the programming model to use io-vector-oriented APIs with *Zbuf* for zero-copy, and co-design with LUNA's flow control to adopt the inlined-r2c thread model.

The upper bound of LUNA. In this paper, we discussed that LUNA can efficiently utilize 50Gpbs NICs. However, for even higher bandwidth (e.g., 200 to 400Gbps), the LUNA's run-to-completion with a shallow buffer may lead to NIC queue overflow and packets dropping. Moreover, when the message size is 4KB, LUNA needs at least 8 cores to saturate the 100Gbps network bandwidth. Therefore, for adopting a high linkspeed network, we believe leveraging hardware acceleration becomes necessary. Additionally, while TCP can use multi-path transmission with Multipath TCP, the head-of-line blocking problem in TCP and its limitations in failure-recovery have still led us to design a new protocol specifically for high-performance cloud storage. Our recent effort, called Solar [29], which involves using a new transport layer protocol co-designed with the DPU exemplifies this point.

10 Conclusion

In this paper, we describe LUNA, a user-space TCP stack at Alibaba Cloud storage network. We discuss our efforts in building LUNA with a focus on the thread, memory and traffic model. Apart from introducing LUNA, we have also covered various design tradeoffs and lessons from the last five years of development. We hope the experiences shared in this paper shall benefit practitioners from both academia and industry.

Acknowledgments

The authors thank our shepherd Yizhou Shan and the anonymous reviewers for their feedback. We also thank the EBS, Pangu, AIS Fushionnet, OSS and OTS teams for their tremendous help on the LUNA project. This research was partly supported by Alibaba Innovation Research, Alibaba Research Fellow and NSFC(62102424) program.

References

[1] F-stack. http://www.f-stack.org/.

[2] Intel DPDK. https://www.dpdk.org/.

[3] Aliyun. EBS product. https://www.aliyun.com/product/disk.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: minimal near-optimal datacenter transport. In D. M. Chiu, J. Wang, P. Barford, and S. Seshan, editors, ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013, pages 435–446. ACM, 2013. https://doi.org/10.1145/2486001.2486031.

[5] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel, R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill. Empowering azure storage with RDMA. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 49–67, 2023.

[6] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan. Attack of the killer microseconds. Commun. ACM, 60(4):48–54, 2017. https://doi.org/10.1145/3015146.

[7] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014., pages 49–65, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[8] H.-C. Chiang, Y.-P. Dai, and C.-Y. Wang. Full hardware based tcp/ip traffic offload engine (toe) device and the method thereof, Jan. 12 2010. US Patent 7,647,416.

[9] Cisco. VPP. https://fd.io/gettingstarted/technology/.

[10] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021, pages 621–637, 2021. https://doi.org/10.1145/3477132.3483571.

[11] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostic. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In A. Gavrilovska and E. Zadok, editors, 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020, pages 673–689. USENIX Association, 2020. https://www.usenix.org/conference/atc20/presentation/farshin.

[12] S. Floyd, T. Henderson, and A. Gurtov. The newreno modification to tcp's fast recovery algorithm. Technical report, 2004.

[13] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. phost: distributed near-optimal datacenter transport over commodity network fabric. In F. Huici and G. Bianchi, editors, Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT 2015, Heidelberg, Germany, December 1-4, 2015, pages 1:1–1:12. ACM, 2015. https://doi.org/10.1145/2716281.2836086.

[14] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In J. Mickens and R. Teixeira, editors, 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021, pages 519–533. USENIX Association, 2021. https://www.usenix.org/conference/nsdi21/presentation/gao.

[15] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity ethernet at scale. In M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, editors, Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016, pages 202–215. ACM, 2016. https://doi.org/10.1145/2934872.2934908.

[16] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017, pages 29–42. ACM, 2017. https://doi.org/10.1145/3098822.3098825.

[17] Y. Huang, J. Geng, D. Lin, B. Wang, J. Li, R. Ling, and D. Li. LOS: A high performance and compatible user-level network operating system. In Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017, pages 50–56, 2017. https://doi.org/10.1145/3106989.3106997.

[18] Intel. Flow-bifurcation. https://doc.dpdk.org/guides-18.08/howto/flow_bifurcation.html.

[19] Intel. Intel 5800x. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html.

[20] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014, pages 489–502, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[21] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019., pages 345–360, 2019. https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[22] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks. Profiling a warehouse-scale computer. In D. T. Marr and D. H. Albonesi, editors, Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015, pages 158–169. ACM, 2015. https://doi.org/10.1145/2749469.2750392.

[23] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. E. Anderson. TAS: TCP acceleration as an OS service. In Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019, pages 24:1–24:16, 2019. https://doi.org/10.1145/3302424.3303985.

[24] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. Socksdirect: datacenter sockets can be fast and compatible. In Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019, pages 90–103, 2019.

[25] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, H. Wang, S. Liu, L. Chen, Z. Wu, H. Qiu, D. Liu, G. Tian, C. Han, S. Liu, Y. Wu, Z. Luo, Y. Shao, J. Wu, Z. Cao, Z. Wu, J. Zhu, J. Wu, J. Shu, and J. Wu. More than capacity: Performance-oriented evolution of pangu in alibaba. In 21st USENIX Conference on File and Storage Technologies (FAST 23), pages 331–346, 2023.

[26] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: high precision congestion control. In J. Wu and W. Hall, editors, Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019, pages 44–58. ACM, 2019. https://doi.org/10.1145/3341302.3342085.

[27] 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. E. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019, pages 399–413, 2019.

[28] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgment options. Technical report, 1996.

[29] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang, R. Liu, C. Shi, B. Fu, J. Zhu, J. Wu, D. Cai, and H. H. Liu. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In F. Kuipers and A. Orda, editors, SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022, pages 753–766. ACM, 2022. https://doi.org/10.1145/3544216.3544238.

[30] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In S. Gorinsky and J. Tapolcai, editors, Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 221–235. ACM, 2018. https://doi.org/10.1145/3230543.3230564.

[31] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. Acceltcp: Accelerating network applications with stateful TCP offloading. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 77–92, Santa Clara, CA, Feb. 2020. USENIX Association.

[32] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In 16th USENIX Symposium on Networked

Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019, pages 361–378, 2019. https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[33] J. Ousterhout. It's time to replace tcp in the datacenter. arXiv preprint arXiv:2210.00714, 2022.

[34] J. Postel et al. Transmission control protocol. 1981.

[35] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017, pages 325–341, 2017. https://doi.org/10.1145/3132747.3132780.

[36] Samsung. Samsung z-nand ssd. https://semiconductor.samsung.com/ssd/z-ssd/.

[37] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter. Flextoe: Flexible TCP offload with fine-grained parallelism. In A. Phanishayee and V. Sekar, editors, 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022, pages 87–102. USENIX Association, 2022. https://www.usenix.org/conference/nsdi22/presentation/shashidhara.

[38] Z. Wu and H. Chen. Design and implementation of TCP/IP offload engine system over gigabit ethernet. In Proceedings of the 15th International Conference On Computer Communications and Networks, ICCCN 2006, October 9-11, 2006, Arlington, Virginia, USA, pages 245–250. IEEE, 2006. https://doi.org/10.1109/ICCCN.2006.286280.

RubbleDB: CPU-Efficient Replication with NVMe-oF

Haoyu Li[1], Sheng Jiang[1], Chen Chen[1], Ashwini Raina[2], Xingyu Zhu[1], Changxu Luo[1], and Asaf Cidon[1]

[1]*Columbia University*, [2]*Princeton University*

Abstract

Due to the need to perform expensive background compaction operations, the CPU is often a performance bottleneck of persistent key-value stores. In the case of replicated storage systems, which contain multiple identical copies of the data, we make the observation that CPU can be traded off for spare network bandwidth. Compactions can be executed only once, on one of the nodes, and the already-compacted data can be shipped to the other nodes' disks, saving them significant CPU time. In order to further drive down total CPU consumption, the file replication protocol can leverage NVMe-oF, a networked storage protocol that can offload the network and storage datapaths entirely to the NIC, requiring zero involvement from the target node's CPU. However, since NVMe-oF is a one-sided protocol, if used naively, it can easily cause data corruption or data loss at the target nodes.

We design RubbleDB, the first key-value store that takes advantage of NVMe-oF for efficient replication. RubbleDB introduces several novel design mechanisms that address the challenges of using NVMe-oF for replicated data, including pre-allocation of static files, a novel file metadata mapping mechanism, and a new method that enforces the order of applying version edits across replicas. These ideas can be applied to other settings beyond key-value stores, such as distributed file and backup systems. We implement RubbleDB on top of RocksDB and show it provides consistent CPU savings and increases throughput by up to 1.9× and reduces tail latency by up to 93.4% for write-heavy workloads, compared to replicated key-value stores, such as ZippyDB, which conduct compactions on all replica nodes.

1 Introduction

To provide high availability, disk-based key value stores are often replicated on multiple machines [2, 21, 22, 25, 35, 43]. A standard architecture for replicating key-value stores is that each machine runs a local key-value instance, and a replication layer controls which replica gets shipped to each instance.

However, key-value stores spend a significant portion of their computing resources on background compaction operations, which rebalance and garbage-collect the data on disk. For example, in the case of log-structured merge trees (LSM trees), the standard disk-based key-value store design [2, 4, 8, 22, 41], previous work has shown that compaction can consume up to 45% of CPU in production workloads, and by avoiding compaction, key-value stores can increase their throughput by up to 2× [12]. We have reproduced these experiments and find that with RocksDB, compaction consumes up to 72% of the total CPU cycles.

This leads to the simple observation that, in the case of replicated key-value stores, where each node sees identical commands, the compaction operations conducted on each machine that stores the replica of the data represent *redundant effort*. Therefore, we can design an architecture, where a *primary* node conducts compaction operations locally, and then ships the already-compacted files to the *secondary* nodes that store the data copies, thereby significantly reducing their CPU consumption.

However, such an approach has two important drawbacks. First, it increases the amount of network traffic because not only do the regular operations need to be replicated, but also the compacted files. Fortunately, network traffic in modern datacenters is often underutilized; for example, cluster traces from Alibaba [1] and Snowflake [47] show that 50-–75% of network capacity consistently remain idle. Therefore, reducing CPU consumption at the expense of additional network traffic is often a desirable trade-off. Second, shipping the files from the primary to the secondary nodes still requires some processing from both: at the extreme, if both ends use TCP, then shipping the files will incur the cost of processing the TCP packets on both ends, as well as the cost of traversing the storage stack on the secondary nodes.

To address the second problem, we turn to NVMe-oF, a networked storage protocol that minimizes CPU costs at secondary nodes. NVMe-oF extends the NVMe protocol to allow one server to access a disk of a remote server directly, with minimal involvement of the remote server's CPU. Even better, most commodity datacenter NICs support offloading the entire NVMe processing at the remote server, by allow-

ing the remote NIC to talk directly to the NVMe storage device. Therefore, if we use NIC-offloaded NVMe-oF, the secondary's host CPU will not be involved at all in processing the incoming replicated files, thereby completely eliminating all of its CPU costs due to compaction.

However, using NVMe-oF to replicate files across storage nodes creates two challenges. First, since the remote node's local file system (e.g., ext4) is not involved in writing the files, it is not aware of the updated file and its location, has no way to read it, and may even accidentally overwrite it. Second, the key-value application running on the remote node must also be synchronized with the incoming files. Its application-level in-memory data structures must be updated to find and read data from new files that were updated on its local storage device, and it must not read data from stale files that were deleted in the compaction process.

In this work, we introduce RubbleDB, the first distributed storage system that leverages offloaded NVMe-oF for efficient replication. The key contributions underlying RubbleDB's design are mechanisms that provide both *file system synchronization* and *application synchronization* at the remote node, so it can safely and correctly read data that was written to it via NVMe-oF.

In order to simplify file system synchronization, we make the observation that modern SSD-based datacenter storage systems [4, 8, 9, 16, 35] write data in large immutable (and often fixed-sized) chunks, and do not allow in-place updates. Therefore, RubbleDB pre-allocates all on-disk data on all nodes as fixed-sized fixed-location files. RubbleDB maintains a *file map* that stores the mappings between the file names and the pre-allocated file locations, and indicates whether a file contains live or stale data. When a new file is replicated, it is sent to a pre-allocated location that does not contain a live file. When a file is deleted in the compaction process, it is simply marked as stale in the map, and is not actually deleted.

For application-level synchronization, RubbleDB needs to keep the secondaries' in-memory data structures synchronized, so when they read data from disk, they read the most up-to-date object versions. To do so, RubbleDB ensures that changes made to the in-memory data structures in the secondary nodes will be consistent with the compactions executed by the primary node. It also carefully synchronizes the deletion of objects flushed from disk or memory, in order to avoid accidentally deleting objects that were processed out-of-order in the secondary nodes.

Our evaluation demonstrates that RubbleDB consistently leads to significant CPU and I/O bandwidth savings compared to a baseline, which represents the architecture of systems such as Meta's ZippyDB [17, 43] or CockroachDB [42], which run compaction on all nodes in a replication group. These savings enable RubbleDB to consistently achieve the same or higher throughput than the baseline across the entire YCSB suite [20], as well as on five traces from Twitter's key-value cache clusters [49]. In particular, RubbleDB provides a speedup up to 1.9× and a tail latency improvement of up to 93.4%. We also show that RubbleDB consistently provides higher performance in different scenarios, including different replication factors, different numbers of RocksDB instances per physical server, and different types of storage devices.

While in this paper we focus on the particular use case of a replicated key-value store, we believe our design ideas are applicable to other common storage applications with primary-backup replication, such as replicated file systems [16, 26, 34, 48] and disaster recovery and backup services [38].

2 Background and Motivation

This section lays out the background and motivation for the paper. §2.1 provides background information on the most common data structure for disk-based key value stores, the log structure merge tree (LSM tree), and demonstrates that background compaction operations in LSM trees consume significant CPU. §2.2 provides a primer on the NVMe-oF protocol and then shows the performance benefit of using NVMe-oF for storage replication with a microbenchmark.

2.1 The High Cost of Compactions

LSM trees. LSM trees [37] are a popular data structure for disk-based key-value stores, which powers many modern key-value stores, such as RocksDB [8], LevelDB [4] and WiredTiger [9]. Since small random writes significantly hurt SSD (and HDD) performance, the main design goal behind LSM trees is that data written to disk is always written in large contiguous chunks and is never updated in-place.

As a representative system for LSM trees, we provide a primer on how RocksDB, a popular key-value store works. In RocksDB, to avoid small random writes to disk, all incoming data writes are batched in memory, in a data structured called the MemTable. Each entry in the MemTable has a sequence number that enables key versioning. MemTables can be *active*, which means that they are mutable and can be updated with new incoming updates, and *immutable*, which means they are waiting to be flushed and cannot be updated further. Eventually, the immutable MemTables get flushed to disk and written using a format called sorted string table (SST) files, which are composed of sorted key-value pairs. SST files are composed of blocks, each of which can be a data block or a metadata block. The metadata blocks include index blocks whose entries point to the keys at the start of each data block.

SST files are organized hierarchically into levels (L0, L1, ..., LN), where the "upper levels" (e.g., L0 is "higher" than L1 in the hierarchy) store the more recently updated versions of each key-value pair. Data from the MemTable is flushed into L0, which stores files with overlapping key ranges, while the files in lower levels (L1,...,LN) have non-overlapping key ranges.

A key feature of LSM tree-backed stores is *background compaction*, which periodically scans multiple SST files from two adjacent levels, combines them into a single file, and

flushes the new file into the lower level. In this process, deleted and overwritten keys are discarded, freeing up space for new data. Compactions are necessary not only for freeing up space on disk, but also for reducing the number of I/Os required on average to read data from the LSM tree [37].

To reconstruct the LSM tree after a failure, RocksDB persists a log containing changes to the tree, e.g., deletion or generation of SST files. RocksDB records such changes using *version edits*, where a version represents the current set of SST files in the tree. For example, a version edit may record the removal of stale SST files and the generation of new merged files. Although compaction jobs run in parallel, they produce version edits in a serializable order because RocksDB protects the tree status with a mutex.

CPU consumption of compactions. Compactions are expensive and can affect the performance of the key-value store. A compaction job requires reading the data of all the files involved in the compaction (often involving tens of MB of data or more), sorting them, and writing them back to disk.

As an example, we measure the CPU time consumed by compactions by running a microbenchmark (described in §5.2) on a replicated 3-node key-value store, where each node conducts compaction locally, under a data ingestion microbenchmark (YCSB load [20]). In this workload, 72% of CPU time was dedicated solely for compaction jobs! Due to their high cost of compactions, there is a large body of work on reducing their resource consumption in single-node LSM trees [12, 13, 29, 32, 39], e.g., by delaying them, synchronizing them with incoming requests, or optimizing the LSM tree data structures and parameters to reduce their cost.

Saving compaction CPU and I/O bandwidth in replicated key-value stores. Our focus is orthogonal to these single-store optimizations: we make the observation that in settings where the same data is replicated on a set of R key-value stores, we do not have to run R identical compaction jobs across all nodes, which are essentially performing the same exact computation. Therefore, compaction can occur only once (on the *primary* node), and the already-compacted SST files can be shipped to the *secondary* nodes, which hold the backup copy of the data.

Such an approach has the potential to significantly reduce CPU consumption on the secondary nodes, since they no longer need to issue read and write I/O and sort the compacted data, the latter of which typically consumes the most CPU during compaction jobs [12, 13, 29]. In addition, this would eliminate the compaction read I/O of secondary nodes, since they would not need to read the files that need to be merged by the compaction job, but it would not eliminate the secondary's write I/O, since the new file would still have to be written back to the disk. Finally, it would also reduce the memory pressure on the secondary nodes due to compaction.

However, executing compactions only on primary nodes has a price. The primary cost of this approach is increased network bandwidth and NIC resource consumption, since now not only the "regular" incoming read/write requests need to be replicated, but also the post-compaction SST files. Fortunately, in many datacenters the network is often underutilized: for example, in traces from Alibaba [1] and Snowflake [47], 50–75% of the network capacity is idle. In addition, the primary node would consume some additional CPU in shipping the files to the secondary nodes' disks.

Therefore, since this approach involves a trade-off primarily between minimizing CPU consumption on the secondary nodes and increasing total network bandwidth, we seek to ship the SST files with a protocol that will minimize CPU usage on the secondary nodes. To this end, we turn to NVMe-oF, a state-of-the-art networked storage protocol supported by Linux and modern NICs, which can be run without the involvement of the secondary nodes' CPU.

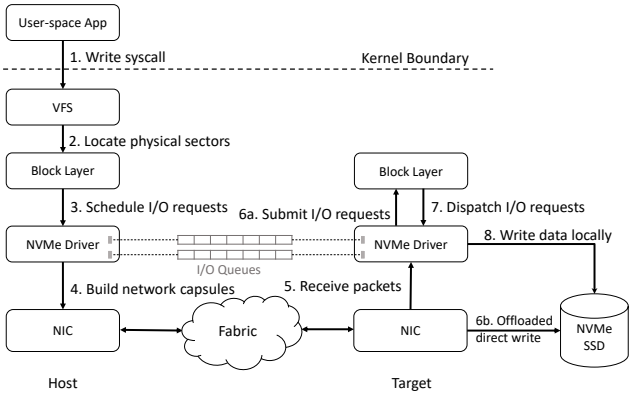


Figure 1: NVMe-oF overview.

2.2 Motivation for Using NVMe-oF

NVMe-oF primer. NVMe-oF is an extension of the NVMe protocol for networked storage. NVMe-oF allows an application to directly access a storage device that is connected to a remote server, using the NVMe protocol. Figure 1 depicts the flow of an NVMe-oF request. The *host* (left side of the diagram) is the server that initiates the request, and the *target* is the remote server and the SSD connected to it. The NVMe-oF request is initiated by an application on the host, which issues a system call, and subsequently traverses the entire OS storage stack, treating it as a regular local NVMe request, until it reaches the NVMe driver.

Take a write request as an example (Figure 1): the userspace application issues a WRITE() system call on the file located on an NVMe-oF mounted disk (step 1), then just like a normal local I/O, it goes through the Linux Virtual File System (VFS) to find the inode, which maps the physical sectors on the disk and is then submitted to the block layer (step 2) where it gets batched by the I/O scheduler, and is dispatched to the host-side NVMe driver (step 3).

The *host* and *target* drivers maintain multiple I/O queues for exchanging the NVMe-oF capsule, which is a data structure that contains essential information needed for an NVMe

	gRPC + WRITE()	NVMe/TCP	NVMe/RDMA
Throughput	1028 MB/s	2986 MB/s	3748 MB/s
CPU	155%	135%	50%

Table 1: Comparison of throughput and CPU consumption of 1 MB writes with different protocols. NVMe-oF (via TCP or RDMA) is much more efficient than replicating through userspace.

communication between the host and the target. The NVMe driver handles this request by constructing a corresponding NVMe-oF command within a capsule, mapping data and metadata from the memory, and submitting it to one I/O queue. The capsule is then forwarded to the relevant network stack (step 4) depending on the fabric type (TCP, RDMA, etc.) and is then forwarded to the target. For NVMe/TCP, the capsule is embedded in TCP packets and contains both data and metadata, while for NVMe/RDMA, the target and the host exchange the capsule using two-sided RDMA operations. With NVMe/RDMA, the capsule records the memory address of the data buffer in the host and the target consequently reads that portion of memory using a one-sided RDMA read.

On the target (step 5), after the driver extracts the NVMe-oF command and user data from the network packet, it generates the block layer request and submits it to the block layer for I/O scheduling (step 6a). The target's NVMe driver, at last, receives the I/O request from the block layer (step 7) and writes the user's data to the local NVMe SSD through the PCIe bus (step 8).

In the past few years, major NIC model lines (e.g., NVIDIA ConnectX, Broadcom Stingray, Intel IPU) have supported completely offloading the NVMe-oF target datapath to the NIC, and allowing the NIC to directly write the data to the NVMe device. This offers an alternate datapath that bypasses the target's CPU completely (step 6b). When the NIC attached to the target receives an NVMe capsule from the *host*, it executes the NVMe request and directly writes data on the NVMe SSD via DMA.

Potential benefit of NVMe-oF for replication. Popular distributed storage systems (e.g., CockroachDB [42] and Ceph [48]) often use an RPC (e.g., gRPC [3]) to send data from the primary to the secondary node, which in turn is written locally to the SSD (e.g., with a WRITE() system call).

We compare the throughput and CPU usage of this userspace-based baseline with two NVMe-oF protocols (NVMe/RDMA, which stands for NVMe-oF over RDMA and NVMe/TCP, which stands for NVMe-oF over TCP), in a microbenchmark that writes 1 MB data chunks over the network in a closed loop, with two servers using the same experimental setup on CloudLab [40] described in §5.1. In the experiment, each server contains one primary node that is writing to a secondary node on the second server, with a total of 256 available cores. The aggregate results are shown in Table 1. The result shows that the throughput of gRPC with WRITE() is only 34% of the throughput NVMe/TCP while the CPU usage is 20% higher. In addition to the more complex logic in the RPC framework, the userspace stack requires extra user-kernel boundary crossings and context switches when the data buffer is delivered to the userspace application from the TCP/IP stack in the kernel and then written to the local file which incurs a kernel trap. NVMe/TCP, on the other hand, processes the data write completely in the NVMe driver in the kernel, therefore saving a substantial amount of CPU cycles in each write request, thereby increasing the throughput. In addition, NVMe/RDMA outperforms NVMe/TCP due to the elimination of unnecessary copying and CPU bypassing.

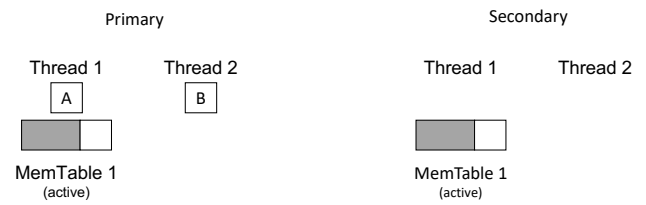
3 Challenges

Substituting a userspace replication protocol with NVMe-oF introduces challenges at two different layers: at the file system level and the application level.

File system inconsistency. NVMe-oF introduces inconsistency at the file system level. A naive way to ship files through NVMeoF is to simply allocate a new file on the remote disk and write to it. However, in such a scheme, the secondary node will not even see the new SST files in its file system. This is because the SST files are created in the primary's file system, and NVMe-oF only forwards NVMe commands, which get executed below the file system layer in the secondary node's storage stack (see Figure 1). So, the primary and secondary nodes may see different files systems on the same NVMe disk. Even worse, the data sent by the host could accidentally overwrite data in physical blocks at the secondary that it is not supposed to access, since the local file system of the target may have changed its file-to-block mapping.

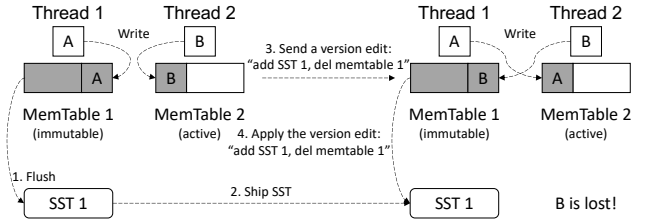
Application inconsistency. Even if the target's file system is synchronized with the host's view, NVMe-oF introduces inconsistency at the application level. Since the persistent key-value store maintains in-memory data structures (e.g., to buffer writes), these data structures may not be synchronized between the primary and the secondary, leading to data loss. In particular, in RocksDB, there will be discrepancies between the primary and secondary node within their MemTables, which store the values of recently-written data in memory.

Figure 2 shows an example where discrepancies in the primary and secondary's MemTables cause data loss in the secondary. Consider the case where there is one active MemTable (MemTable 1), which is nearly full and only has capacity for one more object (Figure 2a). Now consider that two objects (A and B) arrive concurrently. Both primary and secondary use two threads to process incoming requests, and in this case RocksDB does not provide any guarantee on the order that the writes will be processed. In the primary, object A is written before B, and is therefore written to MemTable 1, which is sealed and marked inactive, while object B is written to the newly active MemTable 2. Next, the primary forwards objects A and B to the secondary, but the secondary applies them in the opposite order due to non-deterministic thread scheduling: B is written to MemTable 1, and A is written to MemTable 2. Consequently, the secondary's MemTable 1 stores different



(a) Initial state, single active MemTable.

(b) The secondary executes concurrent requests at a different order.



(c) The secondary removes MemTable 1, losing object B.

Figure 2: An example of inconsistency across node MemTables.

data than the primary's MemTable 1 (Figure 2b).

Now, the primary flushes MemTable 1 to disk, causing it to delete the objects stored in MemTable 1 from memory. If it then ships the new SST file to the secondary, and instructs it to also delete it to delete MemTable 1 as well, this will result in the loss of B at the secondary, because B will not be stored neither in its MemTables, nor on its disk (Figure 2c). In this case, the reason for the data loss is due to the fact that thread scheduling across the nodes in a non-deterministic fashion, so operations are applied in a different order, causing discrepancies.

Making matters worse, even if we had a way to force secondary nodes to process requests in the same order as the primary, the content of the MemTables would still diverge. This is because RocksDB's MemTables store their data using randomized skip lists, which will cause MemTables in different nodes to contain a different number of entries and become full at different times.

4 Design and Implementation

We present the design and implementation of RubbleDB, and explain the key mechanisms that allow RubbleDB to address the inconsistencies introduced by replication via NVMe-oF.

RubbleDB is a replicated key-value store, composed of a set of RocksDB instances, with a replication layer on top. RubbleDB uses chain replication [45] to provide strong consistency and fast recovery. The client only communicates with the replicator layer, which is in charge of dispatching requests to the proper primary node (in case of write) or tail node (in case of a read) and of handling failure recovery. Figure 3

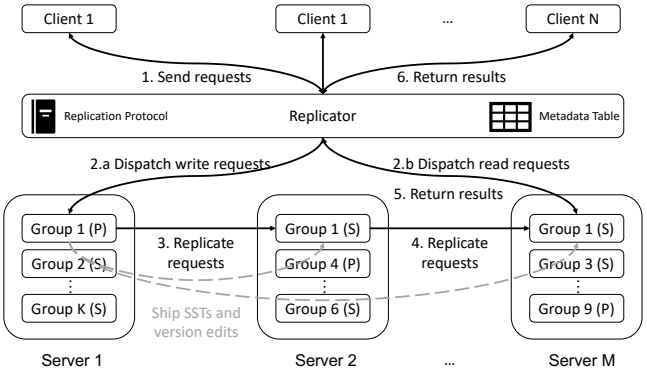


Figure 3: RubbleDB overview.

depicts the overall architecture of RubbleDB. There are N clients and K replication groups, and in between sits the replicator layer. Replication groups contain R RocksDB instances or *nodes*, one of which is the primary, and the others are secondaries. Only the primary performs flush or compaction jobs. Therefore, in addition to replicating client write requests, the primary node also ships compacted SST files via NVMe-oF, assuming sufficient network bandwidth is available. If the network becomes congested, RubbleDB can fall back to local compaction on all replicas. Specifically, RubbleDB compares the latencies of shipping SST files and local compaction. If the former is consistently greater over a time period, RubbleDB falls back to regular compaction. Different replication groups store disjoint key spaces. By default, the R replicas are stored on R different random servers. In the future, we plan to support other more sophisticated data placement policies [18, 19]. We intentionally keep each replication group small (by default 10 GB), so the recovery load can be spread across multiple nodes in the cluster when a server or disk fails. It is worth noting that we assume no dishonest or malicious node (e.g., we assume all nodes operate under a single organization in a single data center). Next, we discuss the design details of the two main key components of RubbleDB: the replicator layer and replication groups.

4.1 Replicator Layer

To provide a clean key-value interface from users and hide the complexity of dealing with the replication protocol, RubbleDB uses a replicator layer as a proxy layer between users and replication groups. Users simply send regular RocksDB requests to and receive results from the replicator layer, which transparently handles the replication protocol. The replicator thus has two roles: 1) routing requests to a replica of the group that contains the requested key-value pairs and 2) detecting and recovering from any failed replicas.

Different replication groups contain separate key spaces. To route requests, the replicator maintains a metadata table that records the key space and network addresses for each replica group. Once it receives a request, the replicator first looks up the group number in the metadata table. Next, according to the replication protocol, it forwards the request to a specific

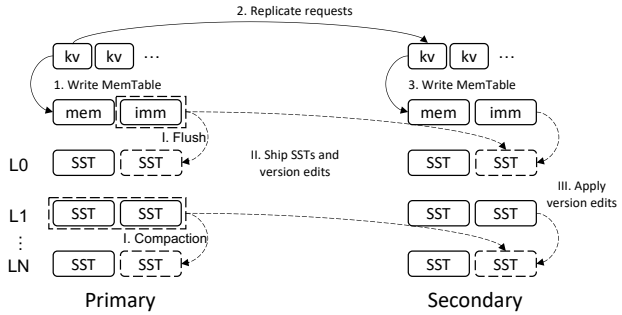


Figure 4: Replication process.

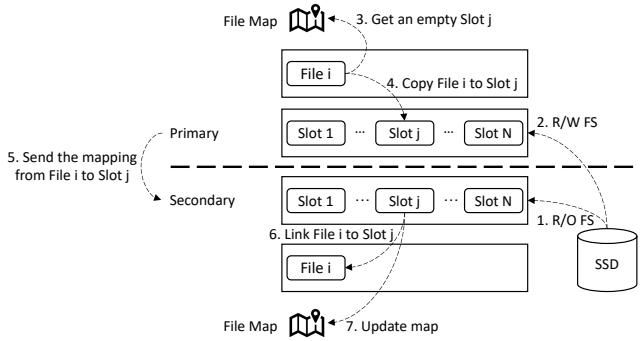


Figure 5: Primary ships SST files to pre-allocated slots.

replica within that group. The replicator also sends heart beat messages to every replica periodically to confirm its health. If it does not receive any replies from a replica after a time threshold, the replica is assumed to have failed, and the replicator starts the recovery process.

In Figure 3 foreground data flows are represented by a solid arrow, while background flows are dashed. The figure only shows the background requests belonging to replication group 1, which is replicated across servers 1, 2 and M. Clients first send requests to the replicator (step 1), who after consulting the metadata table forwards the requests to replication group 1 (step 2). Following the chain replication protocol [45], write requests (e.g., put and update) go to the head (step 2.a), while reads (e.g., get and scan) go to the tail (step 2.b). In the case of writes, the primary (head) replicates the write request to the next secondary in the chain (step 3), which applies the write and then replicates it to the next node in the chain (step 4). When the tail node completes a request (read or write), it will reply to the replicator (step 5), which finally returns the results to the client (step 6).

It is important to note that the replicator is only a logically centralized component that orchestrates traffic and recovery. To prevent the replicator from being a performance bottleneck or a single point of failure, it can be implemented as a distributed fault-tolerant cluster [17, 42]. We leave this direction, as well as other aspects of the replicator's design, such as dynamic load balancing and dynamic key-space partitioning, for future work.

4.2 Replication Groups

Each node within a replication group is a small RocksDB instance, composed of a primary node (head of the chain) and a chain of secondary nodes, which store the backup copies of the data. Figure 4 presents how a primary interacts with one of its secondary nodes. Solid and dashed arrows represent foreground and background operations, respectively. Write requests are executed from the head replica (the primary) to the tail (steps 1-3). Read requests are omitted in Figure 4 because they are only sent to the tail secondary node.

Steps I-III show how RubbleDB avoids background compaction jobs in secondaries. In step I, flush and compaction jobs happen normally in the primary (triggered by filled MemTables or upper-layer SST files). These jobs change the primary's LSM tree in three ways: 1) deletion of the data being compacted (both in-memory immutable MemTables and on-disk SSTs, depicted with dashed rectangles in Figure 4), 2) creation of compacted SSTs (dashed rounded rectangles), and 3) modification of the LSM tree version (information recording current SST files in the tree). RubbleDB ensures that the same changes also occur in the secondaries by shipping both compacted SSTs and version edits over the network (step II). Shipping the compacted SST file addresses 2), so the secondary only needs to delete the original obsolete SST files according to the version edits and update its own LSM tree version (step III).

However, it is not trivial to guarantee the correctness of steps II and III due to the challenges described in §3. In §4.2.1 we discuss how RubbleDB solves the challenge of file system inconsistency, while in §4.2.2 we describe how RubbleDB addresses application inconsistency.

4.2.1 File Pre-allocation

As the primary and secondary nodes mount their own local file systems (e.g., ext4) on top of the same storage device, each local file system will not be aware of changes made by the other file system, e.g., file creation. To ensure that shipped SST files are visible to secondary nodes, RubbleDB uses *file pre-allocation*. Before running, secondary nodes allocate many pre-allocated file slots, which we call a *file pool* on their local storage devices, after which, the primary mounts these devices. So both sides will be aware of the file pool in their local file systems. During runtime, the primary ships an SST file to a secondary by writing the content to a fixed-sized slot in the pool with direct I/O (to make sure the file gets written to disk and bypasses the primary's local buffer cache). Thus, only the data blocks of the slot file are updated and the inode remains unchanged. The secondary can also read the content with direct I/O after the file is written.

Note that this means that secondary and primary nodes cannot rely on the buffer cache to cache hot data blocks from disk. Fortunately, RocksDB (and most other key-value stores) implements its own userspace-based cache, the block cache, which can replace the operating system's buffer cache.

There are four practical issues with this pre-allocation

scheme: 1) determiming the size of slot files, 2) managing slot files in the pool, 3) avoiding dynamic file remapping by the local file systems, and 4) ensuring that RocksDB will correctly point to the pre-allocated files even when it changes file names. We discuss each issue below.

File size. To guarantee that the primary can find a slot to ship SST, secondaries need to allocate a sufficient number of file slots for every possible file size. Fortunately, key-value stores like RocksDB typically store data in more or less fixed-sized (or size-capped) files. Moreover, the number of SST files in each layer of an LSM tree is also limited by compaction. For example, by default in RocksDB, the size of an SST file is 64 MB and the maximum number of SST files is 4448. In this case, a secondary would need to create 4448 64 MB file slots. In our implementation we use a fixed-size file that is slightly larger (17 MB) than the target file size of RubbleDB's RocksDB instances (16 MB), since files may occasionally exceed the target size. When files are smaller than the fixed size of the slot, the remainder of the slot is zero-padded.

Slot management. The primary acquires slots in the pool before shipping SST files to secondaries. Similarly, when deleting an SST file post compaction, the corresponding slot is released. We design a *file map* to track the mapping between slots and SST files and to indicate whether a slot contains a live SST file. Both the primary and secondary nodes have a copy of the map. It is necessary for secondaries to own a map copy because once the primary fails, one of them will be chosen as the new primary.

In a flush or compaction job, the primary first acquires empty slots in its file map and then executes the compaction. After shipping the compacted files to the secondary nodes, it sends the map updates to all secondaries with the version edits, so the same updates are applied in all the secondary nodes. After receiving the updates, the secondary marks the slots of the old files, whose space can be overwritten, as released, and it updates the primary's file map to notify it about the slot release. The reason slots are released by secondary nodes is to avoid the case where the primary node releases a slot, and then acquires it again before the secondary node was notified of the slot release, which would be viewed by the secondary as an illegal operation, where a new file overwrites an already-acquired slot.

File remapping. The pre-allocated file slots' mappings from file offset to physical block address may change over time. Various reasons can cause remapping, including dynamic volume management, file system extent adjustment, etc.. To minimize interference from the file system and volume management, RubbleDB uses a dedicated and static disk partition for the file pool in each secondary node. The partition is mounted as read-only in the secondary, since the secondary never writes to its SSD drive, and read-write in the primary node. In case of a crash, where a secondary needs to become a primary, it remounts with read-write mode.

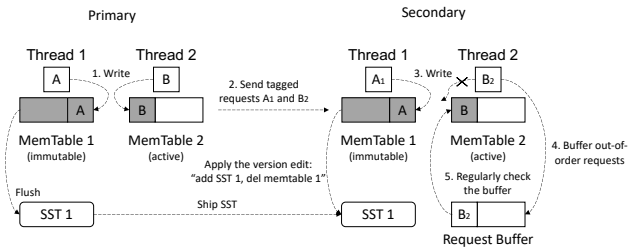


Figure 6: Partially-ordered writes using MemTable ID.

Renaming RocksDB names each SST file with a unique integer, e.g., `002023.sst`. This leads to another issue of pre-allocating slot files: their fixed file names. Because the secondary mounts the partition as read-only, it cannot rename the slot files to the RocksDB format. To address this issue, RubbleDB creates a symbolic link from a file with the RocksDB-defined name to the slot file, so the RocksDB instance on secondary can correctly access its read-only file pool.

Multiple groups. If there are multiple replication groups, primary nodes from different groups will acquire slots concurrently. To avoid contention, RubbleDB creates a dedicated SST pool (and map) for each group. Recall that since each pool sits on a different disk partition, there are no concurrent writers to a file in RubbleDB.

Figure 5 summarizes the file replication workflow. Before the replication group is formed, a disk partition is created for the file pool on each secondary. The primary and secondary mount the partition as read-only (step 1) and read-write (step 2), respectively. Suppose that the primary node generates SST file i in a flush job, it first queries the file map for an empty slot j to ship the SST file (step 3). Next, the content of SST file i is written in slot j with direct I/O (Step 4). The data of slot j will be transferred to the secondary node's SSD via NVMe-oF. At the end of the flush job, the primary sends the mapping between file i and slot j to the secondary (step 5), so the secondary knows how to create the correct symbolic link (step 6) and update its file map copy (step 7).

4.2.2 LSM Tree Synchronization

Flush and compaction jobs are essentially performing merge sort and do not change the actual state of RocksDB from the client's perspective[1]. These merge sorts contain inputs: MemTables and SST files to be merged in the case of flush and compaction, respectively, while the output is always SST files that will be written to disk. This property implies that the inputs and output of a flush or compaction job must contain the same set of live key-value pairs. Primary nodes naturally satisfy this requirement since they execute compaction locally. However, secondary nodes sometimes have mismatched sets of inputs and output live key-value pairs when applying version edits. Recall from the example in Figure 2, in the secondary node, the input to the flush job (MemTable 1) has

[1]Although stale data will be discarded during compaction jobs, it is already ignored by RocksDB since read requests fetch the most recent data.

different live objects from the output (SST 1). Consequently, the secondary node loses B while redundantly storing two copies of A.

To guarantee the data consistency of secondary nodes, they need to ensure that the inputs and outputs of every version edit contain the same set of live objects before applying it. However, comparing all objects across multiple MemTables or SST files is very costly. Instead, RubbleDB forces a partial order of requests and total order of version edits. These two ordering techniques synchronize the secondary nodes' LSM trees with the primary's. We describe them below.

Partially-ordered writes. Figure 6 describes how RubbleDB addresses the MemTable discrepancy issue discussed in the example in Figure 2 by ordering write requests with MemTable ID. In the primary, after an object is inserted to the active MemTable, each write request is returned with the ID of that MemTable (step 1). The primary tags each write request with this MemTable ID and forwards it to the secondary (step 2, the subscripts are the IDs). With the IDs, the secondary now knows to which MemTable the primary wrote each request. The secondary follows the same order as the primary, by maintaining a request buffer to cache out-of-order requests. For example, even if the secondary scheduled thread 2 before thread 1, it will fail to write B_2 to MemTable 1 as its tag (2) does not match with the MemTable ID (1) (step 3). So, thread 2 will store the request B_2 in the request buffer (step 4). When thread 1 executes, A_1 it will be written to MemTable 1. Therefore, MemTables 1 on both the primary and secondary nodes will have the same set of objects, which will not cause data loss like in Figure 2c. Last, every time RocksDB switches to a new MemTable, each thread in a secondary checks the buffer to execute any request that can be applied correctly to the MemTables, i.e. its tag is equal to the ID of the active MemTable (step 5).

This scheme represents a partial order because secondary nodes only sort write requests belonging to different MemTables. Write requests that have the same MemTable ID as the primary's MemTable have identical tags and can execute in any order. This does not affect the correctness when all updates in a MemTable have unique keys because MemTables (skip lists in RocksDB by default) and flush or compaction jobs (merge sorts) will sort them anyway. However, in the case where there are updates for the same key, as both MemTables and flush or compaction only select the most recent update, the secondary has to maintain the same order among those different updates. RubbleDB achieves such an order by further splitting the key space among threads. For example, all updates of key A will be handled by primary's thread 1 in Figure 6. Then, RubbleDB relies on in-order request delivery (e.g., streaming RPC) to ensure those updates arrives at a single thread of the secondary in the same order.

Totally-ordered version edits. Partially ordering writes only guarantees that the secondary nodes eventually have the same live objects in their MemTables as the primary node. However, due to request buffering, updates applied on the secondary nodes may lag the primary, so the same MemTable ID in a secondary node may have fewer entries than the one in the primary. Such lag introduces challenges when applying version edits in secondaries. Back to Figure 6, suppose that at time t, the version edit `(add SST 1, del MemTable 1)` arrives at the secondary but the request A_1 has not been executed. Applying the version edit at time t may allow the client to read A_1, even if it has not been written. This breaks the consistency guarantee of chain replication, which requires that a client can only read a value after it has received an acknowledgment that the value has been written successfully.

To avoid the scenario above, we have to ensure that the sets of live objects in the inputs and outputs of each version edit in the secondary are the same. We exploit the fact that in RocksDB flush or compaction jobs generate version edits in a serializable order (the current version is protected by a mutex) even though they run in parallel. So, the primary node tags version edits with sequence numbers to indicate their order, and the secondary nodes maintain a counter and a buffer for version edits. The counter is incremented every time the secondary applies a version edit. The secondary checks two conditions before applying an edit: 1) whether the sequence number is equal to the counter and 2) whether its inputs are ready. The latter is checked for flush jobs only, since the inputs of a compaction are always ready if it passes step 1) (i.e. the previous flush or compaction job has finished). A MemTable is ready only when it becomes immutable (full). If either of the two conditions fails, the version edit is cached in the buffer, which is regularly checked by all threads.

With these two ordering techniques, RubbleDB synchronizes the LSM tree state in a replication group and addresses the challenge of application inconsistency.

4.3 Implementation Details

We implement RubbleDB using RocksDB 6.14.0 and gRPC 1.34.0, comprising a total of about 900 and 4000 lines of Java and C++ code, respectively. Each replica in RubbleDB is a RocksDB instance, and different parts of the system communicate with each other using streaming gRPC calls. To simulate concurrent clients, we modify YCSB to issue requests as batches to our replicator in an open loop. We open-source all the code on GitHub[2].

5 Evaluation

We seek to answer four evaluation questions:

Q1: How does RubbleDB's SST file replication affect the CPU, network, and disk I/O usage of RubbleDB? (§5.2)

Q2: How does the replication mechanism of RubbleDB affect its performance under different workloads? (§5.3)

[2]https://github.com/lei-houjyu/RubbleDB

Workload	Composition
YCSB Load	100% inserts
YCSB A	50% Read, 50% Update
YCSB B	95% Read, 5% Update
YCSB C	100% Read
YCSB D	95% Read, 5% Insert
YCSB E	95% Scan, 5% Update
YCSB F	50% Scan, 50% Read Modify Write
YCSB G	100% Update
Twitter Cluster 2	100% Get
Twitter Cluster 15	100% Set
Twitter Cluster 19	75% Get, 25% Set
Twitter Cluster 27	85% Get, 15% Set
Twitter Cluster 31	6% Get, 94% Set

Table 2: Workload Characteristics

	Baseline		RubbleDB	
Time spent	Primary	Secondary	Primary	Secondary
Compaction	979	976	987	0
Requests	376	390	397	401
Total	2723		1786	

(a) Replication factor = 2

	Baseline		RubbleDB	
Time spent	Primary	Secondary	Primary	Secondary
Compaction	1319	2759	1375	0
Requests	535	990	570	1176
Total	5603		3121	

(b) Replication factor = 3

	Baseline		RubbleDB	
Time spent	Primary	Secondary	Primary	Secondary
Compaction	1713	5455	1846	0
Requests	692	1930	745	2254
Total	9790		4845	

(c) Replication factor = 4

Table 3: CPU time (s) breakdown under YCSB load, with a co-location factor of 1 and different replication factors.

Q3: Does the utility of NVMe-oF change as a function of the available storage resources? (§5.3)

Q4: How fast can RubbleDB recover from failures? (§5.4)

5.1 Experimental Setup

Setup. We conduct all experiments on CloudLab [24, 40]. Unless otherwise specified, replication groups run on multiple r6525 servers and clients run on one c6420 machine with the replicator. Each r6525 server has two 32-core AMD 7543 CPUs at 2.8 GHz, 256 GB DDR4 memory, a 1.6 TB Dell Enterprise SSD, and a dual-port Mellanox ConnectX-6 100 Gb NIC. By default, RubbleDB uses the Mellanox NIC's NVMe-oF offload feature. A c6420 server has two 16-core Intel Xeon Gold 6142 CPUs at 2.6 GHz and 384 GB DDR4 Memory. The OS is Ubuntu 20.04 LTS with a Linux version of 5.4.0. We configure NVMe-oF target offloading following NVIDIA's official guide [7].

RocksDB configuration. We intentionally keep each key-value instance small, so that if an instance fails there will be a relatively small amount of data to re-replicate. Therefore, we use 16 MB SST files and MemTables and an L0 of size 64 MB, so the LSM tree will contain 64 GB data at most. Direct I/O is enabled with a 2 GB block cache. We run DB instances on each server within a cgroup with 4 physical cores. The number of background threads is therefore set to 4 (number of cores). All other parameters remain default.

	Baseline		RubbleDB	
	Read	Write	Read	Write
R = 2	163.7	185.6	94.6	206.6
R = 3	241.4	274.4	97.8	309.9
R = 4	343.3	387.5	101.7	410.7

Table 4: The read and write I/O (GB) on one node, with co-location factor of 1 and different replication factors.

	Baseline		RubbleDB	
	gRPC	NVMe-oF	gRPC	NVMe-oF
R = 2	34.5	0	34.5	105.7
R = 3	57.2	0	57.3	211.1
R = 4	80.0	0	80.1	314.7

Table 5: The total network traffic (GB) via gRPC and NVMe-oF on one node, with co-location factor of 1 and different replication factors.

Benchmark. We evaluate RubbleDB on all YCSB [20] workloads and five Twitter production traces [49]. Table 2 summarizes the workloads' read-write ratio. Four clients concurrently access all replication groups.

Baseline. For an apples-to-apples comparison, the baseline is a replicated RocksDB system, which is configured identically to RubbleDB, except that it does not replicate SST files, and does not include the various mechanisms RubbleDB uses to support NVMe-oF replication (e.g., buffering at the secondary nodes, processing version edits in-order). The baseline here would represent the standard approach of replicated key-value stores, such as ZippyDB [17,43] and CockroachDB [42], where each node compacts its data independently.

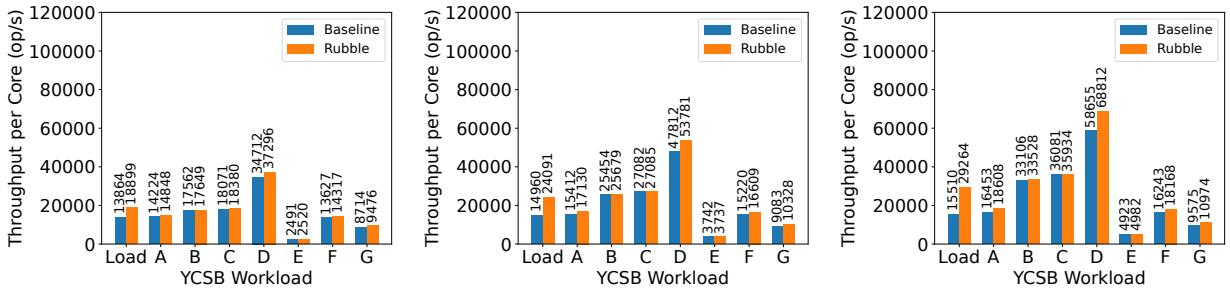
Evaluation metrics and terms. We use two primary evaluation metrics: throughput per core, which represents CPU efficiency, and tail latency. We use two knobs replication factor (R) and co-location factor (C) to indicate the numbers of replication groups (K), servers (M), and replicas in our experiments. We define $C = \frac{K}{M}$ and fix $M = R$, so, $K = C \times R$. For example, a replication factor of 3 and co-location factor of 2 means that on 3 servers ($M = 3$) exist 6 RocksDB instances ($K = 2 \times 3$) (2 primaries and 4 secondaries).

5.2 Performance Breakdown (Q1)

We run the YCSB load workload with a co-location factor of 1 and replication factors of 2, 3, and 4 in this section to collect CPU, disk, and network statistics.

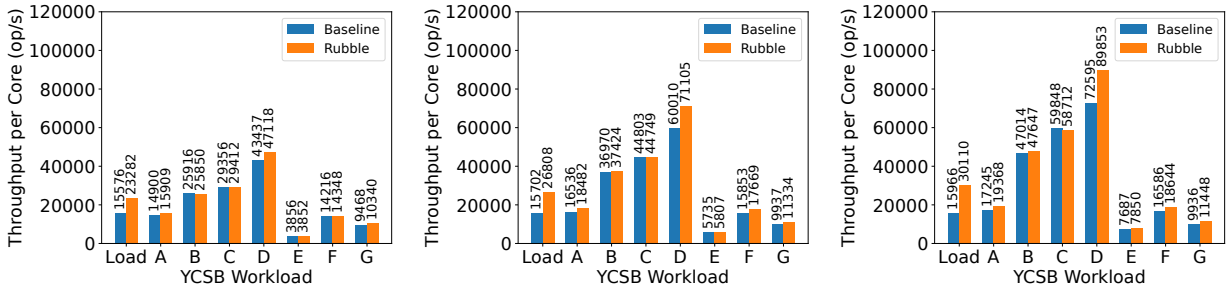
CPU savings. Table 3 presents the amount of CPU time the baseline and RubbleDB spend performing compaction and handling incoming requests. Handling requests includes both reading and writing data from RocksDB, as well as handling the incoming RPCs (i.e. via gRPC), buffering data on the secondary nodes, and applying version edits.

As expected, the secondary nodes on RubbleDB consume no CPU cycles executing compactions, while in the baseline system, each secondary node consumes roughly the same



(a) Replication Factor = 2 **(b)** Replication Factor = 3 **(c)** Replication Factor = 4

Figure 7: YCSB throughput as a function of replication factor with a co-location factor of 1.



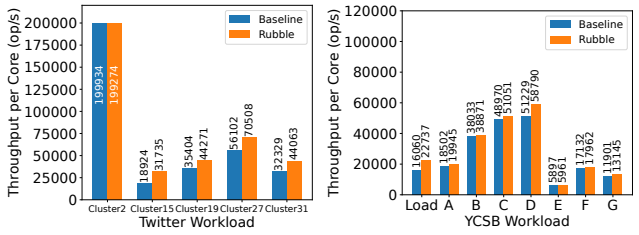
(a) Replication factor = 2. **(b)** Replication factor = 3. **(c)** Replication factor = 4.

Figure 8: YCSB throughput as a function of replication factor with a co-location factor of 2.

amount of CPU cycles as the primary (there are $R-1$ secondary nodes per primary). Under R=2, 3, and 4, the primary node of RubbleDB consumes 0.8%, 4.2%, and 7.8% more compaction CPU than the primary node of the baseline, respectively. This is because the primary has to send compacted SST files and version edits to each secondary node. The overhead increases with the number of secondary nodes.

In terms of handling regular requests, the primary node of RubbleDB consumes slightly more CPU (up to 7.7%) than the baseline's primary node, because it tags every write request with a MemTable ID. The secondary nodes of RubbleDB consume up to 18.8% more CPU than the baseline's, because of the need to buffer incoming requests and version edits. All in all, due to the reduction in the compaction load of the secondary nodes, RubbleDB spends 34.4%, 44.3%, and 50.5% less time processing the same workload than the baseline with R=2, 3, and 4, respectively.

I/O savings. Table 4 reports the amount of data read and written by one node. Since we run the YCSB load workload and disable the write-ahead log, the I/O is caused by compaction. In RubbleDB, only the primary performs compaction, which reads the inputs files and ships compacted SST files to every secondary. Therefore, RubbleDB's read I/O keeps nearly constant, 98.0 GB on average, while its write I/O grows with the replication factor proportionally, averagely $R \times 103.1$ GB. Both the read and write I/O in the baseline, however, increases with the replication factor because all nodes perform compaction. So, RubbleDB saves more read I/O with a higher replication factor, up to 44.2%



(a) **(b)**

Figure 9: Throughput on (a) Twitter cluster traces with a replication factor of 3 and co-location factor of 1, and (b) YCSB using Optane SSD with a replication factor of 2 and co-location factor of 2.

when $R = 4$. There is a modest increase (12.9% at most) in write I/O due to the padding of SST files in RubbleDB, which increases the amount of data that is written for each SST file. We leave reducing the overhead of padding to future work.

Network overhead. Table 5 presents both the gRPC and NVMe-oF traffic. The former consists of forwarding key-value requests and version edits, while the latter includes shipping SST files. The network overhead in RubbleDB includes: (a) sending version edits by gRPC and (b) shipping SST files via NVMe-oF. We approximate (b) by calculating the total volume of shipped SST files. From Table 5, (a) is negligible, and (b) is close to the compaction write I/O.

5.3 End-to-end Performance (Q2, Q3)

Throughput with YCSB. Figures 7 and 8 compare the throughput per core of RubbleDB with the baseline under the load and YCSB workloads, with a co-location factor of 1 and 2, respectively. RubbleDB consistently provides the

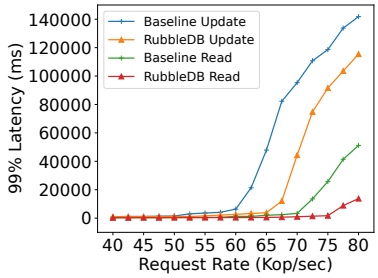


Figure 10: 99% Latency of YCSB A with replication factor of 3 and co-location factor of 1.

same or higher throughput per core compared to the baseline, and has a higher relative speedup for workloads with a high percentage of writes.

As the replication factor increases, RubbleDB provides higher relative gains. For example, under the load workload with a co-location factor of 2, a replication factor of 4 (Figure 7c) yields a speedup of 1.9×, while the speedup of $R = 2$ is 1.5×. The reason is that with higher replication factors, the baseline spends more secondary cores cycles per replication group executing compactions, while RubbleDB experiences a very marginal increase in the primary's CPU consumption (due to the need of shipping SST files to additional secondary nodes). Therefore, with a higher replication factor, RubbleDB has the ability to marshal more available CPU cycles belonging to the freed up secondary node cores, in order to process more incoming requests. In addition, RubbleDB achieves higher absolute throughput and speedup with a co-location factor of 2. The reason is that with more co-located replication groups, RubbleDB is better able to utilize the CPU, since there are more available pending tasks to execute at any given time.

Throughput with Twitter traces. We measure RubbleDB's throughput on five Twitter traces[3] with different read-write ratios, including cluster 2, 15, 19, 27, and 31 [49]. As Figure 9a shows, for write-heavy traces, RubbleDB provides a speedup of 1.7× and 1.4× in clusters 15 and 31, respectively. For cluster 19 and 27, which are ready-dominant, RubbleDB still achieves a 1.3× speedup. These results are largely consistent with the YCSB results.

Tail latency. RubbleDB provides better tail latency than the baseline when there are many compactions. Prior work has shown that compaction jobs interfere with request processing, leading to high tail latencies [12, 13]. Since RubbleDB significantly reduces the overall compaction load, as a result, it decreases the chance that compactions interfere with regular requests.

Figure 10 shows the 99th percentile latency under the YCSB A workload with 3 replicas and a co-location factor of 1, RubbleDB reduces 99th percentile latency of updates and reads by 11.5%-92.1% and 18.4%-93.4%, respectively.

[3]We sample 30GB records from the traces as we have 3 replication groups

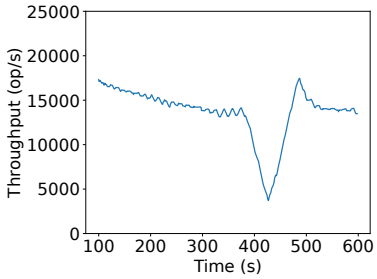


Figure 11: Throughput when a node fails under YCSB load.

The absolute latency is high because our system uses a batch size of 1,000 in evaluation, which means each replica returns a reply after processing all 1,000 requests in a batch. Since write requests go though all replicas sequentially, the update latency will be 3× higher than read latency. We also observe one data point (updates at 40 Kop/sec) showing 14.2% tail latency degradation. This is because out-of-order writes will be cached in the request buffer. Such queuing overhead only appears under light compaction pressure.

Object size. By default, YCSB uses 1 KB objects. When we run YCSB with smaller objects, which are typical in many datacenter settings [11, 17, 49] RubbleDB consistently provides even higher speedups, because a larger fraction of CPU time is spent on compacting data. For example, under YCSB load with a replication factor of 3 and co-location factor of 1, RubbleDB exhibits a 1.6× speedup with 100 B objects compared to a 1.5× speedup with 1 KB objects.

Different storage devices. We try to understand whether a different type of storage device affects RubbleDB's performance. To this end, we run RubbleDB and the baseline on two d750 servers from CloudLab, each of which use Intel Optane SSD P5800X, an SSD with single-digit µs average latencies.We run the experiment with a replication factor of 2 and co-location factor of 2. We are only able to run this experiment with two servers, because of the low availability of Optane SSD on CloudLab.

The results are presented in Figure 9b. Interestingly, the usage of low-latency storage does not materially affect RubbleDB's speedup. While the absolute throughput numbers for read-heavy workloads are higher (for an apples-to-apples comparison compare this experiment with Figure 8a), in the load workload the results are nearly identical. The reason is that while Optane SSD has much better latency than the enterprise SSD we use in the other experiments, its bandwidth is relatively similar, and in the case of LSM trees, write throughput will be determined by disk I/O bandwidth rather than I/O latency, since disk writes are sequential and large. We conclude that RubbleDB provides speedups on very different types of storage devices.

5.4 Recovery Performance (Q4)

To test RubbleDB's recovery from failure, we run a 3-node setup with a single replication group, and kill one of the tail

secondary nodes. We follow the recovery algorithm in [45], which designates the "middle" secondary node as the new tail. We plot the throughput over time in Figure 11. As the figure shows, due to the nature of chain replication RubbleDB is still able to service requests throughout the period when the node is down. In total, it takes about a minute and a half for the cluster to get back to its full throughput capacity.

6 Related Work

We split the related work into two categories: (a) replicated key-value stores, (b) systems that share data with different protocols, e.g., NVMe-oF and RDMA.

Replicated key-value stores. The typical design of replicated key-value stores and databases, such as ZippyDB [17,22, 43], CockroachDB [42], MongoDB [6] and Cassandra [31], is to implement a replication layer on top of multiple single-instance key-value stores, such as RocksDB [8], LevelDB [4] and WiredTiger [9]. In all these systems, all nodes that store up backup copies of data perform their own compactions, leading to high CPU and disk read I/O consumption.

There are several prior systems that do some form of compaction offloading. Ahmad et al. [10] propose offloading large compactions in HBase to a remote compaction server in order to reduce load on the primary nodes serving incoming requests. Hailstorm [14], separates the storage and compute layers, and offloads compaction to nodes that have a low load in a peer-to-peer fashion. Both of these systems allow shifting the computational load of compactions from an overloaded node to an underloaded one, but unlike RubbleDB do not reduce the total compaction load on the cluster by running compaction only once for replicated data.

Closer to RubbleDB, Tebis [46] is a replicated key-value store that reduces CPU consumption by avoiding compacting data multiple times for each replicated chunk of data. However, Tebis has several major design differences from RubbleDB and therefore faces different challenges. First, Tebis' design is based on a key-value architecture that separates keys from values [33]. Therefore, secondaries need to rewrite all the pointers in the indices. Due to the choice of key-value separation, Tebis cannot be applied to standard key-value stores that do not separate keys from values, such as RocksDB, LevelDB or WiredTiger. In addition, while key-value separation provides significant gains with large objects, it can degrade performance for small object workloads, which are common in datacenters [11, 17, 49]. Second, in Tebis, only the primary processes requests, whereas secondary nodes merely store replicated SST files. So, Tebis does not encounter the application inconsistency issue in RubbleDB. Third, instead of NVMe-oF, Tebis uses RDMA with local writes to ship SSTs, which cannot leverage the offloading feature of the NIC. Also, Tebis does not need to deal with inconsistencies caused by the file system.

Storage systems that use NVMe-oF. Several systems use NVMe-oF to access data from remote blocks [5, 15, 27], but only allow each application instance to exclusively access their SSDs. Therefore, these systems do not allow a primary node to replicate to a secondary node's disk directly over NVMe-oF. In other words, unlike RubbleDB, in order to replicate data, these systems require the primary to go through the entire application software stack of the secondary nodes.

Storage systems that use RDMA. Similar to NVMe-oF, the RDMA protocol allows one host to access the other hosts' memory without the CPU involvement of the target. There are a large number of in-memory systems that exploit RDMA for faster operations [23,28,35,44]. While both one-sided RDMA and NVMe-oF may introduce synchronization challenges at the target, the challenges are different, since NVMe-oF operates directly on block storage, potentially introducing corruptions to the local file system at the target.

Shared file systems. Shared file systems [26, 36, 48] provide users across different servers with a consistent view of a file system. However, providing a consistent file system abstraction across multiple nodes can come at a significant performance and scalability cost [30]. Since replicated key-value stores do not require a full synchronized file system interface across nodes, running them over a distributed file system would incur unnecessary overhead.

7 Conclusions

This work explores how to utilize NVMe-oF, a CPU-efficient networked storage protocol, for a common storage use case, replication. The main challenge in using NVMe-oF for replication is that data might need to be read by the target node in parallel to the replication process, introducing inconsistency both at the file system and application level. We demonstrate how such inconsistencies can be addressed in the context of a replicated LSM tree-based key-value storage system, RubbleDB, using two primary mechanisms: file pre-allocation and application data structure synchronization. We believe our ideas can be applied in other common storage settings, such as distributed file systems (e.g., HDFS [16], Ceph [48]) and for storage or application backup. In addition, with the trend of NIC accelerators becoming more powerful in contrast with the plateauing of CPU performance, we anticipate using NVMe-oF for common storage operations will become even more attractive in the future.

8 Acknowledgments

We thank our shepherd, Philippe Bonnet, and our anonymous reviewers for their valuable feedback and suggestions. We also thank Muli Ben-Yehuda, Jianan Luo and Michael J. Freedman for their helpful feedback during the project. This work was supported by NSF award CNS #2106530 and ARO award W911NF-21-1-0078 and was conducted on CloudLab.

References

[1] Alibaba Cluster Trace 2018. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.

[2] Apache Cassandra. http://cassandra.apache.org/.

[3] gRPC. https://grpc.io/.

[4] LevelDB. http://leveldb.org/.

[5] LightBits Labs. https://www.lightbitslabs.com/.

[6] MongoDB. https://www.mongodb.com/.

[7] NVIDIA's NVMe-oF Target Offload Configuration. https://enterprise-support.nvidia.com/s/article/howto-configure-nvme-over-fabrics--nvme-of--target-offload.

[8] RocksDB. https://rocksdb.org/.

[9] WiredTiger Storage Engine. https://www.mongodb.com/docs/manual/core/wiredtiger/.

[10] Muhammad Yousuf Ahmad and Bettina Kemme. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment*, 8(8):850–861, 2015.

[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[12] 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 *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, July 2017. USENIX Association.

[13] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[14] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated compute and storage for distributed LSM-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 301–316, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Tim Bisson, Ke Chen, Changho Choi, Vijay Balakrishnan, and Yang-suk Kee. Crail-KV: A high-performance distributed key-value store leveraging native KV-SSDs over NVMe-oF. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2018.

[16] Dhruba Borthakur. HDFS architecture guide. *Apache Hadoop project*, 53(1-13):2, 2008.

[17] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC 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.

[18] Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gun Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 31–43, 2015.

[19] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, San Jose, CA, 2013.

[20] 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.

[21] 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. *SIGOPS Operating Systems Review*, 41(6):205–220, October 2007.

[22] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage*, 17(4), oct 2021.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet,

Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.

[25] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 21)*, page 440–456, Virtual Event, Germany, 2021.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[27] Daegyu Han and Beomseok Nam. Improving access to HDFS using NVMeoF. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–2. IEEE, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[29] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, 2018.

[30] 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, pages 345–359, New York, NY, USA, 2017. ACM.

[31] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[32] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated key-value storage management for balanced I/O performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 673–687, 2021.

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016.

[34] Arif Merchant. Keynote address II, INFLOW 2014: Optimal flash partitioning for storage workloads in Google's Colossus file system. 2014.

[35] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41, 2011.

[36] Alex Osadzinski. The network file system (nfs). *Computer Standards & Interfaces*, 8(1):45–48, 1988.

[37] 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.

[38] W Curtis Preston. *Backup and Recovery: Inexpensive Backup Solutions for Open Systems*. O'Reilly Media, Inc., 2007.

[39] 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 '17, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *login Usenix Mag.*, 39(6), 2014.

[41] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.

[42] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Amy Tai, Andrew Kryczka, Shobhit O Kanaujia, Kyle Jamieson, Michael J Freedman, and Asaf Cidon. Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 977–992, 2019.

[44] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them

remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48, 2020.

[45] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, volume 4, 2004.

[46] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tebis: index shipping for efficient replication in LSM key-value stores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 85–98, 2022.

[47] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[48] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE 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.

[49] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

Distributed Transactions at Scale in Amazon DynamoDB

Joseph Idziorek, Alex Keyes, Colin Lazier, Somu Perianayagam
Prithvi Ramanathan, James Christopher Sorenson III, Doug Terry, Akshat Vig
dynamodb-paper@amazon.com
Amazon Web Services

Abstract

NoSQL cloud database services are popular for their simple key-value operations, high availability, high scalability, and predictable performance. These characteristics are generally considered to be at odds with support for transactions that permit atomic and serializable updates to partitioned data. This paper explains how transactions were added to Amazon DynamoDB using a timestamp ordering protocol while exploiting the semantics of a key-value store to achieve low latency for both transactional and non-transactional operations. The results of experiments against a production implementation demonstrate that distributed transactions with full ACID properties can be supported without compromising on performance, availability, or scale.

1 Introduction

Application developers have come to rely on database transactions for dealing with failures and concurrency in a distributed system. ACID (atomicity, consistency, isolation, and durability) properties simplify the development process. Transaction atomicity ensures that sequences of operations can be executed without worrying about failures leaving a partial result. Transaction isolation ensures that the developer can write their code without worrying about interference from concurrently executing application instances that read and write shared data.

Despite their utility, NoSQL databases have not generally supported transactions. NoSQL databases such as key-value stores emerged as an alternative to relational databases with a strong emphasis on scalability and performance, especially for customers moving their core data into the cloud. Core features of relational databases, including SQL queries and transactions, were sacrificed to provide automatic partitioning for unlimited scalability, replication for fault-tolerance, and low latency access for predictable performance.

Amazon DynamoDB [9] (not to be confused with Dynamo [8]) powers applications for hundreds of thousands of customers and multiple high-traffic Amazon systems including Alexa, the Amazon.com sites, and all Amazon fulfillment centers. In 2022, over the course of Prime Day, Amazon systems made trillions of calls to the DynamoDB API, and DynamoDB maintained high availability while delivering single-digit millisecond responses and peaking at 105.2 million requests per second. When customers of DynamoDB requested ACID transactions, the challenge was how to integrate transactional operations without sacrificing the defining characteristics of this critical infrastructure service: high scalability, high availability, and predictable performance at scale.

In designing the transaction protocol for DynamoDB, we chose to build transactions differently from other systems and cloud services. The DynamoDB transaction design has the following unique combination of capabilities:

Transactions are submitted as single request. Transactions have commonly been introduced into the database application programming interface (API) with two operations that begin and end a transaction (such as BEGIN and COMMIT in PostgreSQL). These operations serve to delimit the sequence of database operations that are performed within the transaction. The downside of such an abstraction is that there might be a long time between when an application starts a transaction and when it completes its work by committing the transaction. In a multi-tenant service, long-running transactions are undesirable as they tie up system resources on servers that manage data for multiple applications. Instead, DynamoDB transactions comprise a set of operations that are submitted as a single request and either succeed or fail without blocking. Like other DynamoDB operations, transactions provide predictable performance at scale, which is an architectural tenet for DynamoDB.

Transactions rely on a transaction coordinator while non-transaction operations bypass the two-phase coor-

dination. Requiring individual *Gets* and *Puts* to use the full transaction coordination and commit protocol would have had too great of a performance impact on these frequent operations. Thus, all non-transaction operations in DynamoDB are executed directly on the storage servers for the data being accessed, while still being serialized with respect to multi-item transactions.

Transactions update items in place. Increasingly, multi-version concurrency control (MVCC) is employed in database services so that read-only transactions can access old versions of the data while transactions that write data produce new versions. DynamoDB does not support multiple versions of the same item, and adding multi-version concurrency control would have entailed major changes to the storage servers, required version retention policies, and introduced additional storage costs that would need to be passed on to our customers. The implication of a single-version store for transaction processing is that read-only and read-write transactions might conflict.

Transactions do not acquire locks. While two-phase locking is used traditionally to prevent concurrent transactions from reading and writing the same data items, it has drawbacks. Locking restricts concurrency and can lead to deadlocks. Moreover, it requires a recovery mechanism to release locks when an application fails after acquiring locks as part of a transaction but before that transaction commits. To simplify the design and take advantage of low-contention workloads, DynamoDB uses an optimistic concurrency control scheme that avoids locking altogether.

Transactions are serially ordered using timestamps. Techniques for ordering transactions based on timestamps [4] were devised decades ago. The basic idea is that each transaction is assigned a timestamp that defines its position in the serial order. As long as transactions appear to execute at their assigned time, serializability is achieved. A key innovation in the DynamoDB transaction design is extending timestamp ordering to accommodate and exploit the semantics of a key-value store.

This paper presents the DynamoDB transaction API. It also gives examples of how transactions may be used in practice. Furthermore, it illustrates the path of a transaction through the service, describes optimizations to timestamp ordering for workloads with a mix of transactions and singleton operations on a key-value store and, it reports the results of experiments run on a production system, demonstrating predictable performance and scalability.

Operation	Description
`PutItem`	Inserts a new item or replaces an old item with a new item.
`UpdateItem`	Updates an existing item or adds a new item to the table if it doesn't already exist.
`DeleteItem`	Deletes an item from the table
`GetItem`	Reads the item with a given key

Table 1: DynamoDB CRUD APIs for items

2 DynamoDB Application Programming Interface

2.1 Key-value store

DynamoDB [9] is a fully managed NoSQL database service that provides fast and predictable performance at any scale. DynamoDB was motivated by the lessons learned from Dynamo [8] and shares most of the name but little of its architecture. Customers create tables that can grow to virtually any size. A DynamoDB table is a collection of items, and each item is a collection of attributes. Each item is uniquely identified by a primary key. DynamoDB provides a simple interface to store or retrieve items from a table or an index.

2.2 Read and write operations

Table 1 contains the primary operations available to clients for reading and writing items in DynamoDB tables. Since DynamoDB is a key-value store, the most common operations used by applications are for reading an item (`GetItem`), inserting (`PutItem`), updating (`UpdateItem`), and deleting (`DeleteItem`) an item with a given key. These last three operations are collectively called *writes*. A write operation can optionally specify a condition that must be satisfied to be successful.

2.3 Transactional operations

As shown in Table 2, DynamoDB provides two operations for performing transactions: `TransactGetItems` for read transactions and `TransactWriteItems` for write transactions. These operations are submitted as a single request and either succeed or fail immediately without blocking. `TransactGetItems` and `TransactWriteItems` are executed in a serializable order with respect to other DynamoDB operations.

`TransactGetItems` retrieves the latest versions of items from one or more tables. Since it conceptually reads all of the items at a single point in time, the returned values are from a consistent snapshot. DynamoDB rejects the `TransactGetItems` request if a conflicting

```
//Check if customer exists
Check checkItem = new Check()
        .withTableName("Customers")
        .withKey("CustomerUniqueId")
        .withConditionExpression("attribute_exists(CustomerId)");

//Update status of the item in Products
Update updateItem = new Update()
        .withTableName("Products")
        .withKey("BookUniqueId")
        .withConditionExpression("expected_status" = "IN_STOCK")
        .withUpdateExpression("SET ProductStatus = SOLD");

//Insert the order item in the orders table
Put putItem = new Put()
        .withTableName("Orders")
        .withItem("{"OrderId": "OrderUniqueId", "ProductId" :"BookUniqueId", "CustomerId"
:"CustomerUniqueId", "OrderStatus":"CONFIRMED","OrderCost": 100}")
        .withConditionExpression("attribute_not_exists(OrderId)")

TransactWriteItemsRequest twiReq = new TransactWriteItemsRequest()
            .withTransactItems([checkItem,putItem, updateItem]);

//Single transaction call to DynamoDB
DynamoDBclient.transactWriteItems(twiReq);
```

Listing 1: DynamoDB Write Transaction Example

Operation	Description
`TransactGetItems`	Reads a set of items from a consistent snapshot and returns their values
`TransactWriteItems`	Performs a set of writes that include PutItem, UpdateItem, and DeleteItem operations and optionally a set of conditions
`CheckItem`	Checks that the latest value of an item matches the condition

Table 2: DynamoDB Transaction APIs

operation is in the process of modifying any item being read.

`TransactWriteItems` is a synchronous and idempotent write operation that allows multiple items to be created, deleted, or updated atomically in one or more tables. `TransactWriteItems` uses a client request token to guarantee idempotency. The transaction may optionally include one or more preconditions on current values of the items. DynamoDB rejects the `TransactWriteItems` request if any of the preconditions are not met.

To motivate the need for multi-table write transactions with preconditions, consider an online marketplace application. The application stores data in three DynamoDB tables - `Customers`, `Products`, and `Orders`. Upon registration, every customer receives a unique identifier that is used as a key in the `Customers` table which stores customer information such as customer id, customer billing and shipping address. The `Products` table contains information about the products, such as their price and availability; each product is uniquely identified by its product identifier. Orders are stored in the `Orders` table where each order has a unique identifier. A successful order requires the customer account to be verified, the product to be available and marked as sold, and the order itself to be created. These operations should be performed atomically as a single transaction. Listing 1 gives an example of a transaction that purchases a book. This transaction verifies that the customer account exists without updating any attributes in the `Customers` table using `CheckItem`, verifies the book is in stock, and marks the product as sold in the `Products` table using `UpdateItem`, and creates an entry in the `Orders` table using `PutItem`.

3 Transaction execution

3.1 Transaction routing

All operations sent to DynamoDB reach a fleet of front-end hosts called request routers. Request routers authenticate each request and route the request to the appropriate storage nodes based on the key being accessed. The mapping of key-range to storage nodes is maintained in a

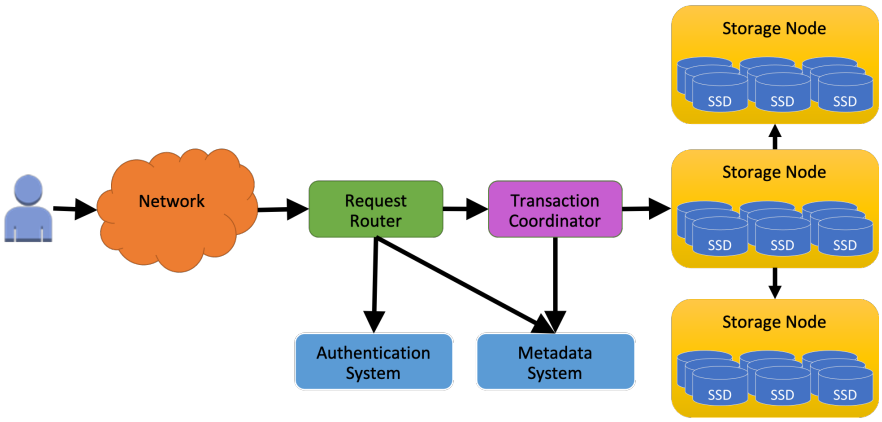


Figure 1: DynamoDB Transactions high-level architecture

metadata subsystem.

Similar to non-transactional requests, each transaction operation initially is received by a request router. The request routers performs the needed authentication and authorization of the request and forwards it to a fleet of transaction coordinators. Any transaction coordinator in the fleet can take responsibility for any transaction. The transaction coordinator breaks the transaction into item-level operations and runs a distributed protocol in which the storage nodes for these items participate. Figure 1 illustrates the high-level diagram of the components involved in the execution of a transaction.

3.2 Timestamp ordering

Timestamp ordering [4, 13] is used to define the logical execution order of transactions. Upon receiving a transaction request, the transaction coordinator assigns a timestamp to the transaction using the value of its current clock. To handle the overall transactions load, there are a large number of transaction coordinators operating in parallel, and different transaction coordinators assign timestamps to different transactions. As long as transactions appear to execute at their assigned time, serializability is achieved.

Once a timestamp has been assigned and preconditions checked, the storage nodes participating in the transaction can perform their portions of the transaction without coordination. Each storage node independently is responsible for ensuring that requests involving its items are executed in the proper order and for rejecting conflicting transactions that cannot be ordered properly.

Although serializability holds even if the transaction coordinators do not have synchronized clocks, more accurate clocks result in more successful transactions and a serialization order that complies with real time. The clocks in the coordinator fleet are sourced from the AWS time-sync service [1], thus keeping them closely in sync (within a few microseconds). However, even with perfectly synchronized clocks, transactions can arrive at storage nodes out-of-order due to message delays in the network, failures and recovery of transaction coordinators, and other system issues. Storage nodes deal with transactions that arrive in any order using stored timestamps.

3.3 Write transaction protocol

A two-phase protocol ensures that all of the writes within a transaction are performed atomically and in the proper order. To achieve atomicity, the transaction coordinator prepares all items in the first phase. In the second phase, if all the storage nodes accept the transaction, then the transaction coordinator commits the transaction and instructs the storage nodes to perform their writes. If any of the storage node cannot accept the transaction, then the transaction coordinator will cancel the transaction. Listing 2 shows the pseudo code for the `TransactWriteItem` protocol.

To implement timestamp ordering for write transactions, DynamoDB records the timestamp of the write operation with every item. All write operations including singleton writes and writes within `TransactWriteItems` update the item timestamp.

Storage nodes also persist per-transaction metadata for each in-flight transaction, including the transaction's identifier and timestamp. This metadata is attached to items that are part of the transaction and remain with the items during partition-related changes, such as split. This ensures that such changes do not interfere with transactions and can happen in parallel. This information about a transaction is updated and checked during the two-phase protocol and can be discarded once the transaction has completed.

In the prepare phase of the protocol, the transaction coordinator sends a message to the primary storage nodes for the items being written. This prepare message in-

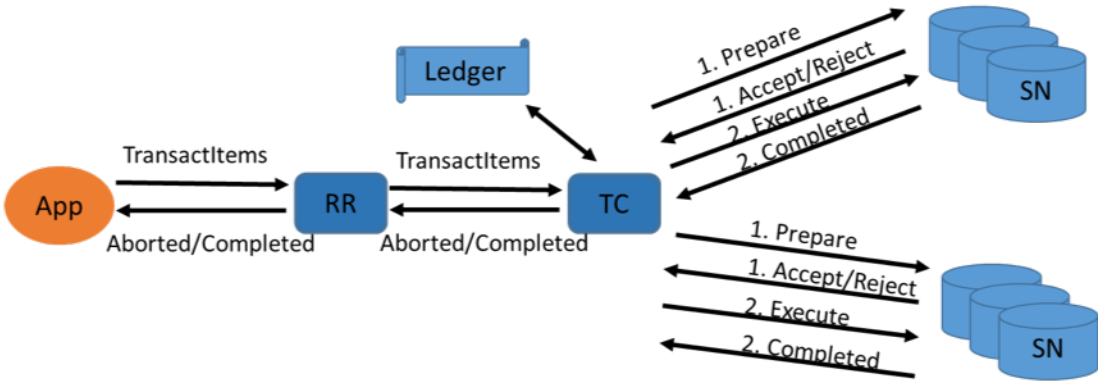


Figure 2: Two-phase protocol

```
TransactWriteItem(TransactWriteItems input):
  #Prepare all items
  TransactionState = PREPARING
  for operation in input:
       sendPrepareAsyncToSN(operation)

  waitForAllPreparesToComplete()

  #Evaluate whether to commit or cancel the
  transaction
  if all prepares succeeded:
     TransactionState = COMMITTING
     for operation in input:
        sendCommitAsyncToSN(operation)
     waitForAllCommitsToComplete()
     TransactionState = COMPLETED
     return SUCCESS
  else:
     TransactionState = CANCELLING
     for operation in input:
        sendCancellationAsyncToSN(operation)
     waitForAllCancellationsToComplete()
     TransactionState = COMPLETED
     return ReasonForCancellation
```

Listing 2: `TransactWriteItem` protocol

cludes the transaction timestamp, transaction ID, and the operation that the transaction intends to perform on the item. The storage node accepts the transaction if all of the following criteria are true for every local item that is part of the transaction:

- All preconditions on the item are met.
- Writing the item would not violate of any of the system restrictions such as exceeding the maximum item size.
- The transaction's timestamp is greater than the item's timestamp indicating when it was last written.
- The set of previously accepted transactions that are attempting to write the same item is empty.

Listing 3 shows the pseudo code for prepare phase of the `TransactWriteItem` protocol. Note that these last two conditions are correct but over restrictive and can be relaxed as discussed in the next section.

If the transaction is accepted by all the participating storage nodes, then the transaction coordinator will commit the transaction. If the transaction is not accepted by any of the participating storage nodes, then the transaction coordinator will cancel the transaction. After the decision has been made to commit the transaction, each participant storage node performs the desired writes on its local items and records the timestamp of the transaction as the items' last write timestamp. Items for which a precondition was checked but that are not being written also have their timestamps updated. Listing 4 shows the pseudo code for commit/cancel phase of the `TransactWriteItem` protocol.

After all participant storage nodes have executed the commit or cancellation, the transaction coordinator responds to the request router with a "completed" message and whether the transaction successfully committed. The request router forwards this response to the customer.

Items that have been deleted require special handling since, once they are deleted, there is no longer a last write timestamp. Instead of maintaining tombstones for deleted items, which would incur both a high storage cost and garbage collection cost if items are frequently created and deleted, DynamoDB stores a partition-level max delete timestamp. When an item is deleted, if the deleting transaction's timestamp is greater than the current max delete timestamp, then the max delete timestamp is set to the transaction's timestamp. When a storage node receives a prepare message for a write to a non-existent item, it compares the new transaction's timestamp against the maximum delete timestamp to decide whether to accept or re-

```
def processPrepare(PrepareInput input):
  item = readItem(input)

  if item != NONE:
    if evaluateConditionsOnItem(item, input.conditions)
      AND evaluateSystemRestrictions(item, input)
      AND item.timestamp < input.timestamp
      AND item.ongoingTransactions == NONE:
        item.ongoingTransaction = input.transactionId
        return SUCCESS
    else:
        return FAILED
  else: #item does not exist
    item = new Item(input.item)
    if evaluateConditionsOnItem(input.conditions)
      AND evaluateSystemRestrictions(input)
      AND partition.maxDeleteTimestamp < input.timestamp:
        item.ongoingTransaction = input.transactionId
        return SUCCESS
  return FAILED
```

Listing 3: `TransactWriteItem` protocol - Prepare phase

ject the transaction. Storing the max delete timestamp at a partition level provides a correct and efficient solution. In the current approach, transactions may be cancelled in instances where they would not have been cancelled if tombstones were maintained for deleted items. Though in practice, an insignificant percentage of transactions are cancelled due to the transaction's timestamp being lower than the partition's maximum delete timestamp.

3.4 Read transaction protocol

Read transactions are also performed using a two-phase protocol, though in a different manner from write transactions and from other systems. The standard timestamp ordering scheme maintains a read timestamp on each item. Updating this timestamp for operations in a read transaction would have turned every read into a more costly write operation on persistent, replicated data. To avoid this latency and cost, DynamoDB devised a two-phase writeless protocol for executing read transactions.

In the first phase of the protocol, the transaction coordinator reads all the items that are in the transaction's read-set. If any of these items are currently being written by another transaction, then the read transaction is rejected; otherwise, the read transaction moves to the second phase. In its response to the transaction coordinator, the storage node not only returns the item's value but also its current committed log sequence number (LSN). The current committed LSN of the item is the sequence number of the last write that the storage node performed and acknowledged to the client. The LSN increases monotonically.

In the second phase, the items are read again. If there have been no changes to the items between the two phases, namely the LSNs have not changed, then the read transaction returns successfully with all of the item values that were fetched. In the case where an item has been updated between the two rounds of the protocol, the read transaction is rejected.

In both failure and success cases, the storage node returns the LSN. By doing so, the transaction coordinator is able to redrive another round of reads for all items without having to restart the entire transaction. In the event that the item is being prepared by a write transaction, the storage node simply rejects the read.

3.5 Recovery and fault tolerance

Since DynamoDB automatically recovers from storage node failures, such failures are of no concern to the transaction protocol. If a storage node that is the primary for an item fails, then leadership will fail over to another storage node that is part of that item's replication group. The metadata about transactions that had been accepted by the previous primary node is persistently stored and replicated within the group, and so is immediately available to the new primary. Transaction coordinators when continuing the transaction protocol are not even aware that they may be communicating with a different set of participating storage nodes.

Transaction coordinator failures are of greater concern. Transaction coordinators can fail because of hardware or software issues. To ensure atomicity of transactions and that transactions complete in the face of failures, coordinators maintain a persistent record of each transaction and its outcome in a ledger. A recovery manager peri-

```
def processCommit(CommitInput input):
  item = readItem(input)

  if item == NONE
    OR item.ongoingTransaction != input.transactionId:
    return COMMIT_FAILED

  applyChangeForCommit(item, input.writeOperation)
  item.ongoingTransaction = NONE
  item.timestamp = input.timestamp
  return SUCCESS

def processCancel(CancellationInput input):
   item = readItem(input)

   if item == NONE
    OR item.ongoingTransaction != input.transactionId:
      return CANCELLATION_FAILED

   item.ongoingTransaction = NONE

   #item was only created as part of this transaction
   if item was created during prepare:
       deleteItem(item)

   return SUCCESS
```

Listing 4: `TransactWriteItem` protocol - Commit/Cancel phase

odically scans the ledger looking for transactions that have not yet been completed (and for which a reasonable amount of time has passed since the transaction was received). Such *stalled* transactions are assigned to a new transaction coordinator who resumes executing the transaction protocol. In the case where a transaction coordinator is incorrectly determined to have failed and its transaction reassigned, it is okay for multiple coordinators to be finishing the same transaction at the same time since duplicate attempts to write an item are ignored by its storage node.

When the transaction has been fully processed, a *completed* record is written to the ledger indicating that no further work is required. Information about a transaction can be purged from the ledger when it has been completed, though retaining these records turns out to be useful for monitoring and debugging.

The transaction ledger is a DynamoDB table with transaction identifiers as the key. Multiple recovery managers regularly scan the ledger in parallel for stalled transactions that must be resumed. Each recovery manager starts its scan of the table from a random key and scans up to thousands of transactions.

Storage nodes also invoke recovery when local items have stalled transactions. If a storage node receives a write or read for an item that is already being written by another transaction, then it checks to see if the pending transaction on the item may have stalled. If the accepted transaction has a timestamp that is older than some threshold, the storage node sends a message with the key for the item and the pending transaction id. The recovery manager receiving this message checks the ledger for the state of the transaction and, if the transaction has not been completed, resumes its execution.

4 Adapting timestamp ordering for key-value operations

The classic timestamp ordering concurrency control scheme [4, 13] can be extended with novel optimizations when applied to a key-value store where reads and writes of individual items are mixed with multi-item transactions. Individual key `get` and whole item `put` operations can be added to an ordered execution history, while allowing for increased concurrency and the ability to execute operations out of order. We have implemented some of these techniques in DynamoDB and others we plan to integrate as we hear more feedback from our customers. This section describes our innovations on timestamp ordering along with the benefits.

Reads to individual items can always be performed successfully even if there is a prepared transaction that is attempting to write that item. A `get` operation that is not part of a transaction is routed directly to a storage node that is responsible for the key of the item being read, bypassing transaction coordinators. The contacted storage

node immediately returns the latest stored value regardless of whether a prepared transaction may later overwrite the item. Implicitly, this `get` operation is assigned a read timestamp that is later than the write timestamp on the stored item and before the commit timestamp of the prepared transaction. In other words, the read is serialized between the last completed write and the pending transaction.

Writes to individual items can be performed immediately and serialized before any prepared transactions in many cases. Non-transactional `put` requests are also directly routed to the storage nodes for the item being written. The primary storage node assigns a write timestamp that is earlier than the timestamps of any transactions in the prepared state. Note that a prepared transaction has not yet performed its intended write to the item, and thus it is okay for a received put to jump ahead of such a transaction in the serialization order. The same holds for individual modify and delete operations that are received directly by storage nodes. The outcome of such operations will likely be overwritten by a prepared transaction if and when it commits. There is one case where a single-item write cannot jump ahead of a previously prepared transaction, namely when a condition on the item had been checked during the process of preparing the transaction and the newly received write operation may violate that condition. For example, suppose that a write transaction is attempting to withdraw 100 dollars from a bank account and it includes a pre-condition to ensure that the current balance contains sufficient funds. If this transaction is in the prepared state, and its condition has been verified, then the system cannot allow another withdrawal that reduces the balance below 100 dollars to jump ahead of the prepared transaction. Nor can the system permit the item to be deleted. In general, it is challenging for storage nodes to determine whether a previously checked arbitrary condition might be violated by a newly received write. However, doing so for common conditions, like numerical bounds checking, could substantially reduce rejected write operations in contentious workloads.

Writes to individual items can be performed immediately or delayed and serialized after any prepared transactions in other cases. Even if a newly arriving single item write operation violates a checked condition for a prepared transaction, the storage node need not reject the write. The storage node can buffer the write operation until the transaction completes. Note that an already prepared transaction is expected to commit or cancel quickly. Waiting for the transaction is not likely to add significant delay to new write operations and the added delay is typically less than rejecting the write and requiring the client to resubmit it. Once the transaction is completed, a queued write operation can be assigned a later timestamp and serialized after the transaction. As a further optimization, if the storage node receives a put or delete operation that has no precondition, then this operation can be assigned a write timestamp that is later than that of any previously prepared transactions and can be performed immediately. If and when a prepared transaction with an earlier timestamp commits, its writes will be ignored.

Write transactions can be accepted even with an old timestamp. If a write operation that is part of a transaction arrives at a storage node that has already performed a write (either an individual put or transactional put operation) with a later write timestamp, this transaction can still be accepted and enter the prepared state. If this transaction is committed, its write operation is ignored with the observation that, even if performed earlier, it would have been completely overwritten by the later put operation. This argument does not hold if the last write was a modify operation that partially updated the item's contents. The benefit of accepting a transaction with an old timestamp, although it has no effect on some items being written, is that the transaction may contain write operations on other items that are allowed to complete.

Multiple transactions that write the same item may be prepared at the same time. A storage node that has already prepared a transaction can accept a second transaction that is attempting to write the same item. That is, for any given item, a series of transactions that are writing the item may enter the prepared state before any of those transactions commit and perform their writes. If the transactions contain `put` operations that fully overwrite the item's contents (or `delete` operations), then the transactions can actually commit in any order as long as the `put` (or `delete`) of the transaction with the latest timestamp is the last one to be performed. Transactions with modify operations that perform partial updates must execute in their assigned timestamp order since the final value of the item depends on the sequence of execution.

Read transactions can be executed in a single round rather than using a two-phase protocol. A transaction that reads multiple items could complete in a single phase as follows. Suppose that storage nodes supported a variant of the GetItem operation, called *GetItemWithTimestamp*, that takes a read timestamp as a parameter in addition to a primary key. This *GetItemWithTimestamp* operation returns the latest value of the item if its last write timestamp is earlier than the given read timestamp and if any prepared transactions have later timestamps, and otherwise rejects the request. When presented with a new read transaction, the transaction coordinator assigns a timestamp for the transaction and calls *GetItemWithTimestamp* in parallel for each item that is being read. The coordinator buffers the item values that are fetched. If none of the storage nodes reject the `get` call for having an old timestamp, then the coordinator returns the set of

buffered values as the response to the read transaction call; otherwise, it returns an exception. This approach is optimistic in that a concurrent write to any one of the items being read could cause the transaction to be rejected. While conceptually simple, there is a subtle potential problem with this approach, namely the storage node could later accept a write with a timestamp that is earlier than that of the previously executed read transaction. That could cause a subsequent read-only transaction to not be serializable with respect to a previously executed transaction. This is a well-known issue with timestamp ordering and is avoided by having storage nodes maintain a timestamp recording when each item was last read in addition to the last write timestamp. Storage nodes would then require future write transactions to have timestamps that are later than both the previous read and write timestamps on all items being written.

Transactions that write multiple items in a single partition can be executed in a single round rather than using a two-phase protocol. If all of the items that are being written in a transaction happen to reside in the same partition, and hence are stored on the same storage nodes, then the transaction does not require separate prepare and commit rounds. Since there is only one primary storage node participating in the transaction, it can perform all of the pre-condition checks that are required to accept the transaction and then immediately perform the write operations. The contacted storage node informs the transaction coordinator whether the transaction completed successfully.

5 Experiments

This section presents our findings about the performance of transaction requests along various dimensions, such as request rate, transaction size, and contentious workloads.

5.1 Comparison of latencies for varying throughput of transactions

We conducted an experiment that scaled up the transaction request rate while maintaining the same number of operations per transaction to demonstrate that scale has a minimal effect on the latency of transactions in DynamoDB. There were three workloads in this experiment: one with fifty percent writes and fifty percent reads, one with one hundred percent reads, and one with one hundred percent writes. A uniform key distribution and an item size of 900 bytes were used in these tests. Workloads were scaled from 100 thousand to 1 million operations per second. Note that 1 million operations per second are not same as 1 million transactions per second, as each transaction consists of 3-operations. Figure 3 and Figure 4 shows the 50th and 99th percentile performance of

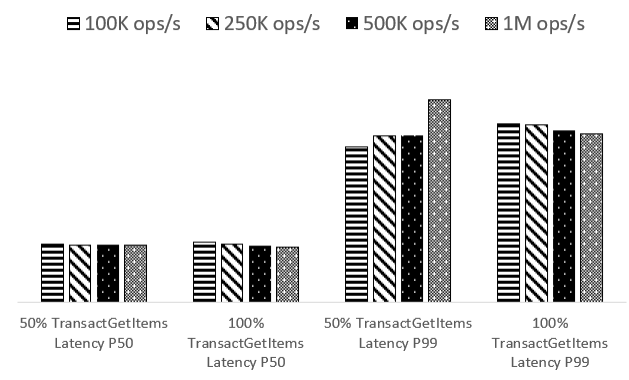


Figure 3: Comparison of `TransactGetItems` latencies for varying throughput

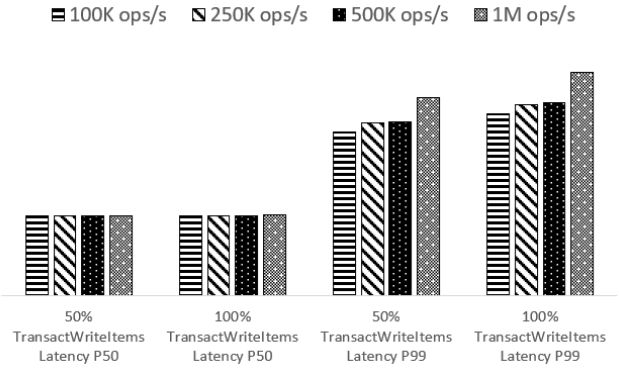


Figure 4: Comparison of `TransactWriteItems` latencies for varying throughput

`TransactGetItems` and `TransactWriteItems` operations for each workload. With the increase in throughput, both `TransactGetItems` and `TransactWriteItems` exhibit negligible variances at P50. The latency increases slightly at P99 as the request rate increases; this is due to increased java garbage collection on the transaction coordinators when the load is heavier.

5.2 Comparison of latencies for varying number of operations per transaction

We conducted an experiment to evaluate the impact of transaction size on performance by varying the number of read and write operations per transaction while maintaining a constant total number of operations. The same uniform key distribution and items of 900 bytes were used as the previous test. Workloads ranged from accessing 3 to 100 items per transaction at a constant rate of 1 million items per second.

Figure 5 and Figure 6 show the performance of the read and write transactions for the various workloads at

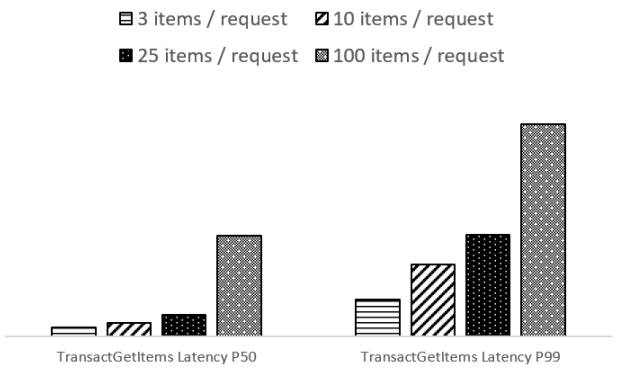


Figure 5: Comparison of latencies for varying number of operations per `TransactGetItems` request

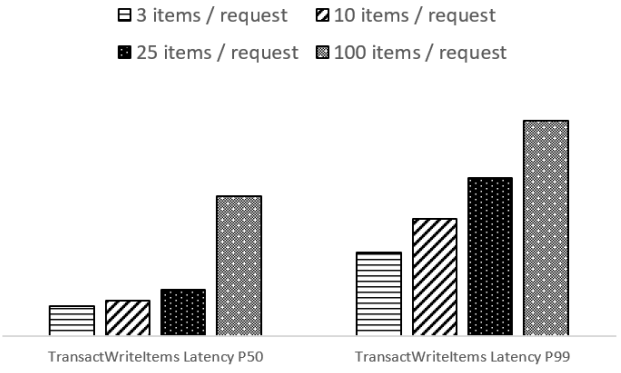


Figure 6: Comparison of latencies for varying number of operations per `TransactWriteItems` request

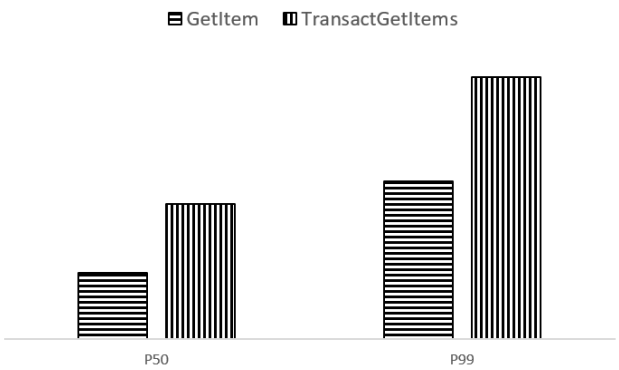


Figure 7: Comparison of latencies for `GetItem` vs single operation `TransactGetItems` request

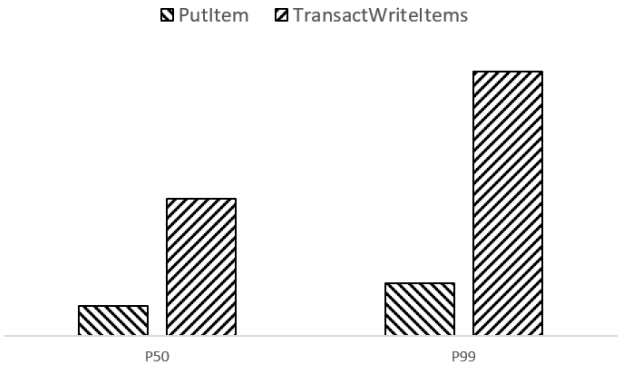


Figure 8: Comparison of latencies for `PutItem` vs single operation `TransactWriteItems` request

the 50th and 99th percentiles. As the number of operations in each transaction increases, so does the latency. Although reads and writes to items within a transaction are processed in parallel, the latency of the transaction request is determined by its slowest operation. Transactions that involve a greater number of operations are more likely to experience a slow read or write. Additionally, the latency of `TransactWriteItems` is determined by the time it takes to persist the request to the ledger. Larger requests take longer to write to the ledger. Also, large transactions result in a larger message payload for the request, which takes longer to travel over the network between the request router and transaction coordinator.

5.3 Comparison of latencies for transactions vs non-transactions

To examine the performance of transactions vs non-transactional requests to DynamoDB, we conducted an experiment comparing the performance of single operation transactional reads and writes against non-transactional singleton reads and writes. For this experiment, we ran tests that submitted 100 thousand requests per second for each of the following DynamoDB APIs; `TransactWriteItems` (transactional write), `TransactGetItems` (transactional read), `PutItem` (singleton non-transactional write), and strongly consistent `GetItem` (singleton non-transactional read). Each request accessed a 900 byte item using the same uniform key distribution that was used in the previous experiment.

Figure 7 shows the performance of single operation transactional vs non-transactional reads at the 50th and 99th percentile. Latency for read transactions is slightly less than 2x the latency for non-transactional reads, on account of the two consistent reads that are required as part of the `TransactGetItems` protocol.

Figure 8 shows the performance of single operation transactional vs non-transactional writes at the 50th and 99th percentiles. Latency for write transactions is about 4x the latency of non-transactional writes. This is as a result of the two-phase write protocol being executed

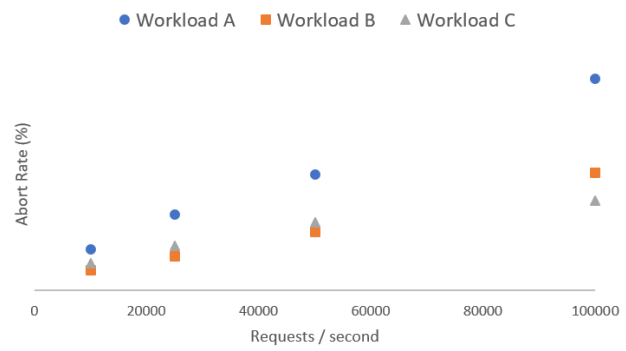


Figure 9: Cancellation rates for workloads with *contention index = 0.001*

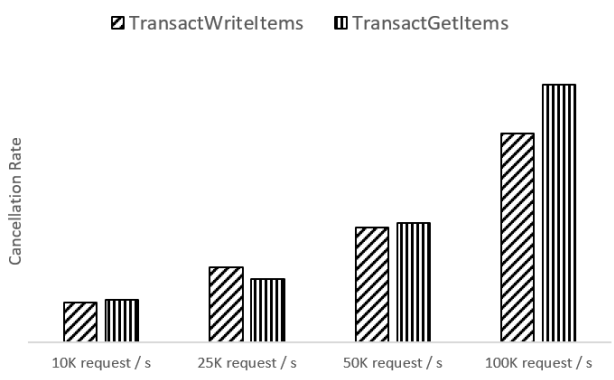


Figure 10: Cancellation rates for workload B with *contention index = 0.001*. Note: each request type represents an equal portion of total traffic

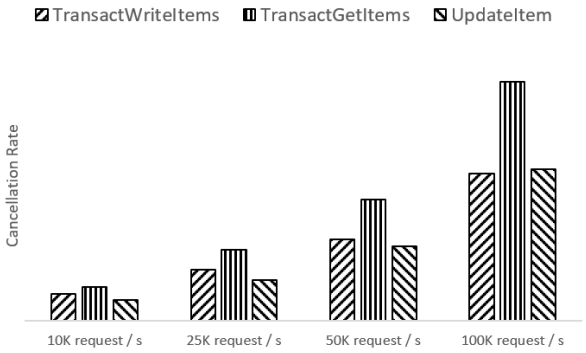


Figure 11: Cancellation rates for workload C with *contention index = 0.001*. Note: each request type represents an equal portion of total traffic

on each `TransactWriteItems` request, with additional overhead being added for writing and checkpointing the transaction state to the Transaction Ledger.

5.4 Comparison of cancellation rate for varied contentious workloads

To examine the performance of transactions on contentious workloads, we ran an experiment with a fixed pool of *hot* items while scaling up throughput. Contention arises when multiple transactions are concurrently trying to access the same items(which are referred to as *hot* items). In this context, a *contention index* [16] refers to the fraction of *hot* items that are accessed by a given transaction. For these experiments, throughput was scaled from 10 thousand to 100 thousand transactions per second with a fixed contention index of 0.001, which indicates that each transaction accesses one of one thousand *hot* items [16]. The experiments ran with three different workloads: workload A consists of write transactions only, workload B consists of 50% write transactions + 50% read transactions, and workload C consists of transactions + non-transactions operations (25% write transactions, 25% read transactions, 25% non-transaction writes, 25% non-transaction consistent reads). Each transaction accesses 10 items with one of the items being from the *hot* item pool and the remaining 9 items being from a much larger set of keys. For non-transaction reads and writes, each item is chosen from the *hot* item pool. For each test, we measured the cancellation rate, which is the percentage of requests that were rejected because of a conflict with another transaction on a given item.

Figure 9 reports the cancellation rates for the workloads with *contention index = 0.001*. For all workloads, the cancellation rate increases with the transaction request rate. As each item can only be acted upon by a single transaction at a time, the level of contention and the cancellation rate rise when more transactions include operations on the pool of *hot* items. Workload A, write transactions only, had the highest rate of cancellations for all transaction requests rates. Comparatively, workload B, with 50% write transactions and 50% read transactions, had about half the cancellation rate as workload A at all levels of throughput. The cancellation rates are lower for workload B as read transactions cannot be the source of conflict.

A `TransactGetItems` operation will be cancelled (rejected) if any item being read has a write transaction in progress or if the item has changed between the two phases, but will not trigger a cancellation of any other operation. Moreover, figure 10 highlights that read and write transactions were cancelled at similar rates for workload B at all throughput levels with both types of transactions only getting cancelled if there was an ongoing write transaction on the targeted item.

Workload C had the lowest cancellation rates at all throughput levels, as a result of having fewer sources of conflict than the other workloads. Figure 11 provides a

breakdown of cancellation rates by operation type for workload C. `GetItem` (non-transaction reads) had no cancellations at all throughput levels as they are serializable with transactions and do not get rejected; if a `GetItem` request is received while a write transaction is in progress on a given item, the `GetItem` will read the current item value without conflict. Both `UpdateItem` (non-transaction writes) and `TransactWriteItems` (write transactions) have comparable cancellation rates as these requests will only be cancelled because of a conflict if there is an ongoing `TransactWriteItems` operation on the targeted items. Finally, `TransactGetItems` (read transactions) had the highest cancellation rate of any operation during this test since read transactions execute optimistically and conflict with any concurrent write.

6 Related work

A growing number of NoSQL databases have added support for transactions in recent years. Each of these systems choose different tradeoffs, resulting in a variety of isolation levels, expressiveness, and relationships with non-transactional writes [3, 5, 12, 14–17].

Many of the systems use a two-phase commit protocol that is similar to DynamoDB's protocol. G-store [7] and L-store [11] are two examples of systems that propose an alternative to two-phase commit protocols. They avoid the two-phase commit protocol by co-locating all the keys of the transaction on the node that processes the transaction and executing the transaction on that single node.

Some systems use locks [2, 16] for concurrency control, while others use timestamps. Different systems use various sources of time, including precise clocks [5], local nodes' clocks, and hybrid logical clocks [10]. Granola [6] is an example of system that uses both locks and timestamps for concurrency control; a transaction is executed either in locking mode or timestamp mode.

7 Conclusion

Adding transactions to DynamoDB without impacting the scale, availability, durability and predictability that customers have come to expect was a daunting task. Instead of the limited form of transactions provided in previous systems, customers asked for full ACID transactions updating multiple items from different partitions of the same table or across different tables. Working backwards from customer scenarios informed us that long running transactions were not required and that the workloads were not highly contentious. We designed transactions as single-shot operations with optimistic concurrency control using timestamp ordering to ensure that transactions are both serializable and scalable. This work shows that transactions implemented in a replicated and partitioned NoSQL database can be achieved with high scalability, high availability, and predictable performance.

8 Acknowledgements

DynamoDB transactions have been greatly influenced by the invaluable feedback of our customers, driving us to innovate on their behalf. We are fortunate to be accompanied by an exceptional team throughout this journey. We express our appreciation to Shawn Bice, Andrew Certain, Raju Gulabani, Amit Gupta, Rishabh Jain, Vaibhav Jain, Nate Riley, Tony Petrossian, Amit Purohit, Julien Ridoux, Rashmi Krishnaiah Setty, Stefano Stefani, Benjamin Wood, Ming-Chuan Wu, and the entire DynamoDB team for their contributions that have been instrumental to the success of this project. We are grateful to the anonymous reviewers and our shepherd, Leonid Ryzhyk, for their invaluable contributions in refining this paper. Special thanks to Chris Andreson, Darcy Jayne, and Murat Demirbas for going the extra mile to provide valuable assistance.

References

[1] Keeping time with amazon time sync service. https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service/.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 159–174, New York, NY, USA, 2007. ACM.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. 2011.

[4] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 285–300. VLDB Endowment, 1980.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle,

S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. 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.

[6] J. Cowling and B. Liskov. Granola:{Low-Overhead} distributed transaction coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 223–235, 2012.

[7] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174, 2010.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, oct 2007.

[9] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, J. C. S. III, S. Sosothikul, D. Terry, and A. Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, Carlsbad, CA, July 2022. USENIX Association.

[10] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems*, pages 17–32. Springer, 2014.

[11] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1659–1674, 2016.

[12] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[13] D. P. Reed. Implementing atomic actions on decentralized data (extended abstract). In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 163–, New York, NY, USA, 1979. ACM.

[14] K. Ren, D. Li, and D. J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11), 2019.

[15] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. 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.

[16] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[17] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, and J. Mulrow. Implementation of cluster-wide logical clock and causal consistency in mongodb. In *Proceedings of the 2019 International Conference on Management of Data*, pages 636–650, 2019.

Prefix Siphoning: Exploiting LSM-Tree Range Filters For Information Disclosure

Adi Kaufman*
Tel Aviv University

Moshik Hershcovitch*
Tel Aviv University & IBM Research

Adam Morrison
Tel Aviv University

Abstract

Key-value stores typically leave access control to the systems for which they act as storage engines. Unfortunately, attackers may circumvent such read access controls via *timing attacks* on the key-value store, which use differences in query response times to glean information about stored data.

To date, key-value store timing attacks have aimed to disclose stored values and have exploited external mechanisms that can be disabled for protection. In this paper, we point out that *key* disclosure is also a security threat—and demonstrate key disclosure timing attacks that exploit mechanisms of the key-value store itself.

We target LSM-tree based key-value stores utilizing *range filters*, which have been recently proposed to optimize LSM-tree range queries. We analyze the impact of the range filters SuRF and prefix Bloom filter on LSM-trees through a security lens, and show that they enable a key disclosure timing attack, which we call *prefix siphoning*. Prefix siphoning successfully leverages benign queries for non-present keys to identify prefixes of actual keys—and in some cases, full keys—in scenarios where brute force searching for keys (via exhaustive enumeration or random guesses) is infeasible.

1 Introduction

Key-value stores serve as the storage engines of many cloud and enterprise systems, from object caches [44,46,47] through stream processing [6,14,54] to database systems [2,29,31,41]. Performance of these modern data intensive systems often depends on their key-value storage engine's performance [51]. Consequently, research on key-value stores overwhelmingly focuses on *efficiency*: from I/O efficiency of writes [20,21], point queries [18,19], and range queries [45,65] to memory efficiency [24,43], energy efficiency [5], multi-core scalability [37,58], and reducing I/O write amplification [51].

But systems also depend on their key-value storage engine for the *security* of stored data. This dependency is not obvious, because key-value stores typically provide only a dictionary abstraction without access control mechanisms [16,30,38,40], leaving access control to the system. Systems enforce access control by mediating user accesses to the key-value store, often based on access control lists (ACLs) stored as value metadata in the key-value store. While this approach blocks users from directly making unauthorized queries, users may still be able to indirectly glean information about restricted data if the key-value store is vulnerable to *timing attacks* [11].

A timing attack exploits differences in query response times to glean information about stored data. A system using a key-value store that is vulnerable to timing attacks can itself become vulnerable to such attacks, because the system's query response time depends on the storage engine's response time, making differences in key-value query response times manifest as differences in the system's response times.

To date, key-value store timing attacks [55,56] have aimed to disclose stored values. We point out, however, that *key* disclosure is also a security threat. In some systems, keys can *explicitly* contain secret data. For example, database systems that use key-value storage engines (e.g., CockroachDB, YugabyteDB, or MyRocks) encode rows (or subsets of rows) onto keys [7,26,28,32]. This makes key disclosure equivalent to database data disclosure. Keys may also be *implicitly* secret, with users expecting them to be hard to obtain. For instance, in object storage systems, such as Amazon S3, identifying valid keys may create an insecure direct object reference vulnerability [48], which enables attackers to probe access to the objects associated with the disclosed keys.

Unfortunately, resilience to timing attacks is not a goal in existing key-value efficiency work—in fact, such resiliency can be at odds with improved performance. In this paper, we demonstrate this trade-off: we analyze key-value store performance mechanisms through a security lens and show that they enable a key disclosure timing attack.

We focus on write-optimized key-value stores based on log-structured merge (LSM) trees [49], which are in widespread use [12,13,15,18,20,22,30,37,42,51,57,60]. In these designs, data in secondary storage consists of multiple immutable files called *SSTables*. LSM-trees can efficiently sustain write-intensive workloads, but queries may require multiple I/Os to search the many SSTables [49,57]. LSM-trees minimize unnecessary I/Os by issuing the I/O only if the queried key is likely to be in the SSTable. Likelihood is determined by querying an in-memory *filter* [10], which space-efficiently *approximately* represents the SSTable's contents. Specifically, filter queries can make "one-sided" errors: if the queried key is present in the SSTable, then the filter always returns true; but for a small fraction of non-present keys, the filter might return a false positive response.

Standard filters can answer point (single-key) queries [8,10,33], but do not support range queries of the form "does the

*Both authors contributed equally to this research.

SSTable contain a key in range $[X,Y]$." Consequently, LSM-tree range queries must search the many SSTables, performing multiple superfluous I/Os [65]. To address this problem, recent work has proposed *range filters*, which are filters that support range queries in addition to point queries. Range filters such as SuRF [65] and RocksDB's prefix Bloom filter (PBF) [25] compactly store some or all prefixes of each of the SSTable's keys, and leverage this information to answer range and point queries.

From a security perspective, however, we show that *certain range filters enable a key disclosure timing attack on LSM-trees*. We describe an attack framework, called *prefix siphoning*, which exploits general range filter characteristics present in both SuRF and PBF. Prefix siphoning successfully leverages benign point queries for non-present keys to identify prefixes of actual keys—and in some cases, full keys—in scenarios where brute force searching for keys (via exhaustive enumeration or random guesses) is infeasible.

Prefix siphoning targets systems with the common design paradigm of storing a key's ACLs as part of its value [1, 4], which means that to check access permissions, the system's query handling always tries to read the queried key's value from the key-value store. Prefix siphoning exploits this property to determine if a random key is one on which the LSM-tree's filter returns a false positive. This is possible because whether the filter returns true or false can be determined by the attacker observing the query's response time, as the filter's response decides whether the LSM-tree performs I/Os. For range filters meeting our characterization, finding a false-positive key implies that the false-positive key shares a prefix with some stored key. Prefix siphoning then performs further point queries—tweaking the queried key—to maximize the length of the disclosed prefix. Prefix siphoning can sometimes subsequently perform a limited enumeration search to fully identify the stored key. Our prefix siphoning implementation performs multiple such steps concurrently, ultimately extracting multiple keys or prefixes.

We evaluate prefix siphoning against SuRF and PBF analytically as well as empirically and demonstrate its feasibility in practice. For example, we successfully use prefix siphoning to extract 64-bit stored keys from a RocksDB [30] datastore employing SuRF in minutes, whereas brute force search of this key space is infeasible. Our analysis and evaluation also quantify the cost of prefix siphoning, showing that it effectively reduces the key search space size by multiple orders of magnitude. For instance, SuRF prefix siphoning requires ≈ 10 M queries to disclose a key from a 50 M 64-bit key dataset—implying a $40992\times$ reduction of the key search space size.

Our results draw attention to the security vs. performance trade-offs in key-value store design, and encourage practitioners and researchers to evaluate the security impact of their work. We hope that our characterization of vulnerable range filters will spur research on more secure filters.

2 Background

This section provides background on key-values stores (§ 2.1), LSM-trees (§ 2.2), and filters (§ 2.3).

2.1 Key-value stores

A key-value store exposes a dictionary-like abstraction with the following operations.

- *put*(k,v). A *put* stores a mapping from key k to value v. If key is already present in the store, its value is updated.
- *get*(k). The *get()* (or point query) returns the value associated with the requested key.
- *range_query*(k_1,k_2). A range query returns all key-value pairs falling within the given range.

Due to their simple and general abstraction as well as high performance, key-value stores serve as the storage engines for many, more complex systems. Examples of such systems include database systems (e.g., Cassandra [42], MongoDB [2], and MySQL [3]) and storage systems (e.g., CEPH [1]).

2.2 LSM-based data stores

The log-structured merge (LSM) tree [49] is a popular choice as the core storage structure for write-optimized key-value stores, which must sustain write-intensive workloads. An LSM-tree consists of levels, each of which contains multiple immutable static sorted table (SSTable) files storing key/value pairs. Two SSTs at the same level never overlap in the key range they store, but SSTables at different levels may overlap.

A *put* request inserts the key-value pair into an in-memory buffer called the Memtable, which is the LSM-tree's only mutable storage object. Once the Memtable fills up, its data is flushed to secondary storage as an SSTable file. The LSM-tree periodically performs *compaction*, where it unifies SSTs between levels to eliminate duplicate (stale) key-value pairs.

A *get* query searches for the target key in a top-down manner: first in the Memtable and subsequently in the relevant SSTable (if it exists) in each level. Searching an SSTable requires I/Os to read it from secondary storage. Once the key is found, its value is returned and the query completes.

However, this design penalizes queries, which may require multiple I/Os to search many SSTables [49, 57]. In particular, a *get()* for a *non-present* key (not associated with any value) searches every level before failing. This not only increases the query response time, but may "thrash" the page cache by reading in many SSTables which will not be accessed later.

LSM-trees minimize unnecessary I/Os by issuing the I/O only if the queried key is likely to be in the SSTable. Likelihood is determined by querying an in-memory filter (described in § 2.3), which space-efficiently *approximately* represents the SSTable's contents. The LSM-tree only reads an SSTable from secondary storage if its filter returns true for

the queried key. As a result, most non-present key queries can respond without performing I/Os.

Likewise, a range filter (§ 2.3.1) can answer both point and range queries with one-sided errors. Using a range filters instead of a standard filter enables an LSM-tree to avoid superfluous I/Os also for range queries, which can improve range query throughput by orders of magnitude [45].

2.3 Filters

A *filter* [10] is a data structure used to approximately represent a set D a of keys. A filter can be *immutable* or *dynamic*. An immutable filter is provided D upon its creation and can subsequently only be queried. A dynamic filter learns D dynamically, via *insert* operations.

Responses for filter queries allow "one-sided" errors: if $k \in D$, then a query for k returns true; but for a fraction of keys $k \notin D$, a query for k might answer true instead of false. We say k is a *positive/negative key* if a filter a query for k answers true or false, respectively. A positive key k is a *false positive* if $k \notin D$. We also say that the filter *passes* positive keys and *rejects* negative keys.

Filters are compared by their space efficiency and false-positive rates. Space efficiency is measured in bits per key. The *false-positive rate* (FPR) of a filter is the probability over keys not in D of being a false positive. I.e., $FPR = FP/(FP+NK)$, where FP is the number of false-positive keys and NK is the number of negative keys. Filters typically have configurable FPRs, with lower FPRs requiring more bits per key for increased accuracy [8, 10, 33].

Bloom filters A Bloom filter [10] is a widely-used dynamic filter (e.g., the default filter of RocksDB). It consists of an m-bit array and j hash functions $H_1, \ldots, H_j$. The parameters m and j determine the filter's FPR and space. Insertion of key k sets the bits indexes $H_1(k), \ldots, H_j(k)$. A query for k returns true if and only if all bit indexes $H_1(k), \ldots, H_j(k)$ are set.

2.3.1 Range filters

A *range filter* is a filter that also supports range queries with one-sided error: a query for $[a,b]$ returns true if there exists $k \in D \cap [a,b]$, but might also return true if $D \cap [a,b]$ is empty.

3 Motivation: avoiding key disclosure

We observe that keys stored in a key-value storage engine can contain sensitive data. It is therefore desirable that users are not able to efficiently discover stored keys that they are not authorized to access. Of course, users can always guess such keys and check if their queries return an authorization error, but such brute force searches are infeasible on large key spaces. The goal is for brute force search to be the only attack option, i.e., to block more efficient key extraction attacks.

Explicitly secret keys Some systems encode secret data in stored keys, which makes key disclosure equivalent to disclosure of the encoded data. For example, database systems such as CockroachDB, YugabyteDB, and MyRocks store table rows as values in a key-value storage engine, with the associated key consisting of the table's id and the row's primary key (one of the cell values). The motivation for this technique is that it enables the database system to perform efficient primary key lookups using key-value store range queries [7, 26, 28, 32].

Implicitly secret keys In many cases, keys are tacitly assumed to be secret or, at least, hard to guess. One example of implicitly secret keys are *object identifiers*. Many web applications and object storage systems maintain object id-to-value mappings in a key-value store. Key disclosure thus allows attackers to probe access to the associated objects, resulting in an insecure direct object reference vulnerability [48]. While objects typically have ACLs, users often neglect to configure these ACLs. This is not a hypothetical concern: for instance, there are numerous scanning tools for "open" (unprotected) Amazon S3 objects [9, 23, 50, 53, 61, 62], and open S3 objects have led to exfiltration of employee information, personal identification information, and other sensitive data [27].

4 Threat model

We consider a *high-level system*, such as a database system or object store, that utilizes a key-value storage engine to respond to user queries. Key ACLs are stored as part of the value associated with the key. As the high-level system performs key-value queries to satisfy a user's query, it checks the ACL of each key it accesses by inspecting the key's value. If the user is not authorized to read a key, the system returns a failure response to the user.

The attacker's goal is to identify keys stored in the system's key-value storage engine. The attacker cannot compromise the system (e.g., to run attack code) and cannot eavesdrop on requests performed by other users and/or on their responses. The attacker can only interact with the system by making requests via its interfaces, such as a representational state transfer (REST) API [34, Chapter 5].

We assume that the attacker can craft their requests in a way that causes the high-level system to make key-value store point queries for arbitrary keys (i.e., chosen by the attacker) while processing the request. For simplicity, we refer to this process as the attacker "querying the key-value store."

We make no assumption about the attacker's physical location with respect to the attacked system. We only assume that the attacker can observe microsecond-level timing differences in the response times of queries for different keys. Prior work has shown that this assumption is true for attacks over both local and wide area networks. For instance, Crosby et al. were able to measure a difference of 20 μs over the circa 2009 Internet (and 100 ns over a local area network) [17]. This ability

can be improved in specific cases. When attacking a system hosted in the public cloud, for example, the attacker can turn themselves into a local-area attacker by placing themselves in the datacenter hosting the target. Moreover, systems that process different requests concurrently (e.g., HTTP/2 servers) are vulnerable to concurrency-based timing attacks [36], which can observe timing differences of 100 ns over the Internet.

5 Prefix siphoning

Prefix siphoning is a general template for conducting timing attacks, extracting partial or full keys, on systems that use an LSM-tree based storage engine with a certain type of *vulnerable* range filter (for both point and range queries). The class of vulnerable range filters contains the filters SuRF [65] and RocksDB's prefix Bloom filter (PBF) [25].

Prefix siphoning exploits range filters that respond to point queries based on key prefix information, which exists to support range queries—i.e., filters where range query support affects the point query implementation. Accordingly, prefix siphoning is based *only on point queries* and does not perform range queries. Henceforth, therefore, the term "query" always refers to a *get()* point query. We leave exploring attacks against range queries to future work.

In the following, we describe the attack's high-level ideas (§ 5.1), characterize the class of vulnerable filters (§ 5.2), and present the attack template (§ 5.3). We describe instantiations of the attack against SuRF and PBF in §§ 6–7.

Notation We treat keys as sequences of symbols over an alphabet Σ (e.g., bytes). When x denotes a key or a set, then $|x|$ refers to the number of symbols or elements, respectively, that x contains.

5.1 High-level ideas

Prefix siphoning exploits an *inherent* trait of filter use in LSM-trees: that whether a key "passes" the filter determines if the LSM-tree searches the SSTable for the key to satisfy a query. This means that for SSTable files that do not reside in the OS page cache, the filter's output for a key significantly affects the LSM-tree's query response time. If the filter returns false for the key, the response is satisfied with only main memory access; otherwise, the LSM-tree needs to perform I/Os to read SSTables from secondary storage. Even for fast storage such as NVMe devices, the difference in query response times between these two cases is enough to affect the system's overall response time in an attacker-measurable way.

This basic filter trait suffices to mount an "approximate membership test" timing attack. The attack simply queries for the target key k and measures the response time. If the response time is fast (i.e., k is rejected by the filters), then k is definitely not stored in the LSM-tree. Otherwise (i.e., k passes some filter), then k is likely in the LSM-tree. The key k might also be a filter false positive and not exist in the LSM-tree, which occurs with a probability bounded by the filter's FPR.

Prefix siphoning starts by randomly generating keys until it finds a key that "passes" the membership test above. For random keys, passing the test overwhelmingly means that the key is a filter false positive. Crucially, it takes just hundreds of attempts to find a false-positive key, because filters are typically configured for FPRs of a few percents for space efficiency reasons [65].

Our main observation is that in vulnerable range filters, a false-positive key likely shares a prefix with some stored key k, whereas negative keys (rejected by the filter) do not (at least with high probability). The crux of a prefix siphoning attack is an algorithm exploiting this trait to *identify* the shared prefix k' through $O(|k|)$ further queries for modified keys iteratively derived from the initial false-positive key.

The revealed prefix of k can already contain sensitive information. But if the system's query responses distinguish between failures due to target key non-presence and lack of authorization, prefix siphoning can fully extract k by performing brute force search of the unknown suffix, thereby *extending* the revealed prefix to k.

Of course, a system whose responses distinguish between non-present and unauthorized keys is also vulnerable to "brute force" key guessing or enumeration attacks based using the above "membership test" primitive. But such attacks are infeasible for many key spaces (e.g., 64-bit or string keys). The point of prefix siphoning is to narrow down the search space by exploiting vulnerable range filters. Moreover, prefix siphoning extracts key prefixes even if the target system's responses do not reveal whether a key is non-present or unauthorized, whereas the "membership test" primitive cannot.

5.2 Vulnerable range filter characterization

We denote an instance of the filter by F and the set of keys it represents by D. A range filter is *vulnerable to prefix siphoning* if it has the following characteristics, denoted C1–C2. They say that a false-positive key κ likely shares a prefix with some key from D and that an attacker can efficiently identify this prefix by making queries for keys derived from κ.

C1 If κ is a false-positive key for F, then with high probability, κ shares a prefix with some $k \in D$.

C2 There exist the following probabilistic algorithms, which work by querying the system:

1. *FindFPK()*: Using an expected constant number of queries, outputs a random false-positive key κ.
2. *IdPrefix(κ):* Given a false-positive κ, uses $O(|\kappa|)$ queries to identify the shared prefix k' that κ shares with some key $k \in D$, if such a prefix exists; otherwise, the output is unspecified.

The *FindFPK* and *IdPrefix* algorithms are specific to the

range filter design, and need to be developed by the attacker.[1] We refer to designing such algorithms for a range filter as *instantiating* the attack against that filter.

C2 implies existence of a timing attack, and is therefore formally sufficient to characterize the vulnerability. In practice, however, our attack instantiations rely on fundamental properties of filter use in LSM-trees. To highlight this aspect of the attacks, we explicitly capture these properties in C3.

C3 1. A *get*(k) query's response time is measurably lower if k misses in every filter than if k hits in some filter.

2. The filter's FPR is small but non-negligible (e.g., 1% or 0.1%).

C3(1) implies that it is possible to distinguish negative from positive keys using query response times. It is trivially true because LSM-trees employ filters to speed up queries for which SSTable searching is superfluous, such as filter misses. Our attacks in this paper exploit microsecond-level time differences between queries satisfied completely from main memory and those that require I/O to secondary storage. (There remain time differences between queries that read an in-memory SSTable residing in the OS page cache and those that do not, due to a filter miss. We leave exploiting such smaller time differences to future work.)

C3(2) implies that generating keys uniformly at random will generate a false-positive key with hundreds to thousands of attempts, on average. It holds because in practice, filters are typically configured with small but non-negligible FPRs (e.g., 0.5%–5%), as negligibly small FPRs blow up the filter's memory consumption [65].[2]

5.3 Prefix siphoning template

Prefix siphoning consists of two phases. First, a preliminary phase learns to distinguish queries of negative and positive keys (§ 5.3.1). The second phase consists of multiple rounds, each of which extracts a key or key prefix (§ 5.3.2). Rounds are run concurrently (see § 9).

5.3.1 Learning to distinguish positive from negative keys

The attack starts with a preliminary phase that builds a distribution of query response times, which is used by the second phase to distinguish positive from negative keys.

The distribution is built by measuring response times of multiple *get()* requests for random keys. With large key spaces, such random keys are mostly negative keys, but a small (though non-negligible) fraction will be positive (due to C3). Such positive keys are overwhelmingly likely to be false positives, but that does not matter for this step, which is only concerned with distinguishing negative from positive keys, regardless of whether the positive output is correct.

The expected distribution observed is a bimodal distribution, with peaks corresponding to the average response time of negative and positive keys. From this distribution, the attacker can derive a cutoff value that likely distinguishes negative (fast) from a positive (slow) queries.

5.3.2 Extracting keys

This phase consists of multiple rounds, each of which extracts a key. Each round consists of three steps: ① finding a false-positive key κ, ② identifying the prefix that κ shares with some stored key k, and, when possible, ③ extending the prefix to extract k. Rounds are run concurrently (§ 9).

Step ① and ② simply invoke the attacker's *FindFPK* and *IdPrefix* algorithms, respectively. These steps are actually the "meat" of the attack, and we later describe their instantiations for SuRF (§ 6) and RocksDB's prefix Bloom filter (§ 7).

Whether step ③ is possible depends on the properties of the attacked system (and this is why it is not part of the vulnerable range filter characterization). If the system's query responses distinguish between failures due to target key absence and lack of authorization, then the attacker can extend the revealed prefix k' with some symbol sequence α and query for the key $k'\alpha$. The response will indicate lack of authorization if and only if $k'\alpha$ is a valid key. The attacker can thus iterate over all possible suffixes until k is found. Because k is not known to the attacker, they must first try all possible single symbol extensions, then all two symbol extensions, and so on. This process requires $O(|\Sigma|^{|k|-|k'|})$ queries, which can be several orders of magnitude less than a full-key brute force search. Crucially, step ③ only attempts to extend "long" prefixes, for which extension is feasible. Other prefixes are discarded.

Rationale for step decomposition For fixed-length keys, it might seem that the *IdPrefix* algorithm (step ②) for identifying the prefix is superfluous. After all, given that κ shares a prefix with some stored key k, the attacker can enumerate all possible suffixes from the end to the beginning, until identifying k. For example, suppose keys are 14-character strings and the attacker has found a false-positive key `manchestercars` because it shares the prefix `manchesterc` with the stored key `manchestercity`. Without knowing (or caring about) the shared prefix, the attacker can start querying for `manchestercara`, `manchestercarb`, ..., `manchestercaaa`, `manchestercaab`, and so on—all of which fail due to key absence—until reaching `manchestercity`, which will fail due to lack of authorization. As before, this process requires $O(|\Sigma|^{|k|-|k'|})$ queries and so it theoretically achieves the same results directly, without requiring an *IdPrefix* algorithm.

Why, then, is existence of an *IdPrefix* algorithm defined as one of the characteristics of a vulnerable filter? The answer is that without knowledge of the prefix, the attacker cannot efficiently schedule their work in step ③. They cannot distinguish a small suffix space (as in the example above) from

[1] Existence of *FindFPK* and *IdPrefix* is required in addition to C1 because a filter satisfying only C1 may not allow an attacker to extract the prefixes.

[2] Prefix siphoning can still be performed for exponentially low false positive rates, but its cost (in terms of number of queries needed) increases proportionally to the decrease in the false positive rate.

a huge space—e.g., if the false-positive key only shared the prefix `m` with `manchestercity`.

The *IdPrefix* algorithm protects us from the above pitfall. By identifying the shared prefix, it enables the attacker to decide whether to try and extend the prefix to a full key. Moreover, when multiple rounds execute concurrently, the attacker can collect many prefixes and then prioritize extending the longest ones.

6 SuRF prefix siphoning

Here, we instantiate a prefix siphoning attack against LSM-trees employing the SuRF [65] range filter. § 6.1 summarizes SuRF and § 6.2 shows that it is vulnerable to prefix siphoning.

6.1 SuRF primer

The succinct range filter (SuRF) [65] is the first proposed general range filter. Like the LSM-tree SSTables it approximates, SuRF is an immutable structure. SuRF can speed up LSM-tree range queries by 5×, but it imposes a modest cost on point queries due to having higher FPRs than a Bloom filter [65].

At a high level, SuRF is a pruned trie. A *trie* is a tree data structure that stores keys sorted according to the lexicographic order of Σ. Each edge is labeled with a symbol and each node corresponds to the concatenation of all edge labels on the path to that node. Each leaf thus corresponds to a key and each internal node to a key prefix (Figure 1(a)). An internal node can also correspond to a key (if the key set is not prefix-free), which is indicated by one of its fields. For space-efficiency, SuRF uses a succinct trie representation.

SuRF further saves space by *pruning* the trie. The basic SuRF variant (SuRF-Base) stores the minimum length key prefixes that uniquely identify each key, i.e., shared key prefixes plus the symbol following the shared prefix of each key (Figure 1(b)). SuRF's pruning results in a space-efficient but only approximate representation of the key set.

Both point and range queries are satisfied from the pruned trie structure. A *get*(k) returns true (possibly erroneously) if and only if the path induced by k terminates at a node associated with a key. For example, in Figure 1(b), `BLOOD` is a false positive. Range queries rely on the trie's ordered structure. For example, to check if the SuRF contains a key $k \in [a,b]$, the query finds the node corresponding to the smallest key $\geq a$. If it corresponds to a key $> b$, the query returns false; otherwise, it returns (possibly erroneously) true.

SuRF variants to reduce FPR SuRF-Base's FPR is data-dependent, i.e., depends on the key set. Compare, for example, two sets of 26 keys: $A = \{x\alpha \,|\, x \in A, \ldots, Z\}$ and $B = \{\alpha x \,|\, x \in A, \ldots, Z\}$, where α is some long string. For A, SuRF's FPR is nearly 100%, as any key except $A, \ldots, Z$ is a false positive. But for B, the FPR is extremely small, as only keys that begin with α pass the filter.

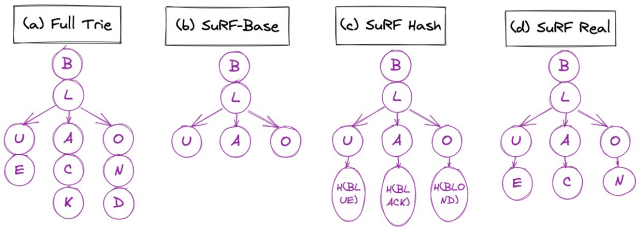


Figure 1: Trie and SuRF variants over the key set `BLUE`, `BLACK`, and `BLOND`. (Figure adapted from [65].)

To improve the FPR, SuRF offers variants that augment SuRF-Base's pruned structure with a few bits per leaf of information about the leaf's suffix. These bits reduce the FPR by allowing queries to reject keys that share a prefix with the stored key but have a different suffix, in exchange for increasing per-key memory consumption.

SuRF-Hash (Figure 1(c)) hashes the leaf's key and stores n bits from the hash value, where n is configurable. *SuRF-Real* (Figure 1(d)) stores the first m bits of the key's suffix, where m is configurable.

6.2 Vulnerability of SuRF

Every SuRF variant has the characteristics defined in § 5.2. C3(1) holds trivially. C3(2) holds empirically: SuRF-Base has an FPR of 4% for random 64-bit keys and SuRF-Hash reduce this FPRs to ≈ 0.1% [65]. C1 holds because in every SuRF variant, *every* false-positive key κ shares a prefix with some stored k—C1 holds with probability 1.

To show that C2 holds, we describe how to efficiently find a false-positive key (§ 6.2.1) and how to identify the prefix that it shares with a stored key (§ 6.2.2). We assume the ability to check if a key is a filter positive or negative key based on measuring query response times. The implementation of this check is described in § 9.

6.2.1 Finding a false-positive key (FindFPK)

For SuRF, our *FindFPK* algorithm simply generates queries for uniformly random keys until it detects a positive response, based on the cutoff determined in the attack's preliminary learning phase (§ 5.3.1). Due to C3, this step is expected to terminate with a few hundreds to thousands of attempts.

We refer to the random positive key found as a false-positive key, because that is the overwhelmingly likely event. However, the attack still works if, unbeknownst to the attacker, the found key is actually a true positive key.

6.2.2 Identifying a shared prefix (IdPrefix)

For a false-positive key κ, let $k = k(\kappa)$ be the stored key whose shared prefix k' with κ is the longest among all stored keys. We write $\kappa = k'\alpha$ and $k = k'\beta$. Our algorithm will output k'.

SuRF-Base/Real To find k', we exploit SuRF's structure, namely that any key starting with a proper prefix of k' is a

negative key. Let $\kappa = \kappa_1 \dots \kappa_n$. We repeatedly remove the last symbol from the key, iteratively checking if the keys $\kappa_1 \dots \kappa_{n-1}, \kappa_1 \dots \kappa_{n-2}, \dots$ are negative or positive keys. These keys will be positive until we remove a symbol from k'. Thus, the key checked before a negative key is found is k'.

If the attacked system does not support variable-length keys, removing symbols is not possible. In this case, instead of removing symbols, we change them. We iteratively check if the keys $\kappa_1 \dots \kappa'_n, \kappa_1 \dots \kappa'_{n-1}\kappa_n \dots$ are negative or positive keys, where $\kappa'_i \neq \kappa_i$. Similarly to before, if the first negative key found is $\kappa_1 \dots \kappa'_j \dots \kappa'_n$ then $k' = \kappa_1 \dots \kappa_j$.

Overall, the number of requests made is $O(|\kappa|)$.

SuRF-Hash SuRF-Hash complicates the attack, because modifying κ's suffix can change its hash value, leading to a key that is rejected by SuRF despite sharing the prefix k'. To address this problem, we assume SuRF's hash function *hash* is public knowledge. (This is a reasonable assumption, because the hash function's purpose is to reduce the FPR and not for security.) We perform essentially the same algorithm(s) as for SuRF-Base/Real, but we only query each modified key κ' if $hash(\kappa') = hash(\kappa)$. We are still essentially assured to find keys to query, because SuRF-Hash stores only a small subset of the hash bits, for space-efficiency reasons. For example, with the recommended 4 hash bits [65] and using 8-bit symbols, on average 1 in 16 symbols tried will yield a hash collision and thus a key usable by the *IdPrefix* algorithm.

Similarly, when trying to extend an identified prefix to a full key (step ③ in § 5.3.2), we can skip querying any candidate key whose hash does not match the false-positive key's hash.

7 Prefix Bloom filter prefix siphoning

This section instantiates prefix siphoning against LSM-trees using the prefix Bloom filter (PBF) [25]. We describe the PBF in § 7.1 and show its vulnerability in § 7.2.

7.1 Prefix Bloom filter primer

The PBF is a Bloom filter-based range filter that supports range queries for ranges expressible as fixed-prefix queries. While PBFs do not provide general range queries, they are currently deployed in real-world key-value stores such as RocksDB [30] and LittleTable [52].

A PBF consists of a Bloom filter and a predetermined prefix length, l. When a key k is inserted into the PBF, both k and its l-bit prefix are inserted into the Bloom filter.

PBF range queries must be for ranges of the form "all keys starting with α," where α is an l-bit string. They are answered by querying the Bloom filter for α. If this query responds false, the dataset does not contain keys within the target range.

The PBF answers point queries by querying the Bloom filter for the queried key. We remark that if the high-level system does not prioritize point query efficiency, the PBF can be configured to only store key prefixes. In this case, the PBF implements a point query for key k by querying its Bloom filter for k's l-bit prefix. This option reduces the PBF's memory consumption but increases the FPR of point queries. This PBF configuration does not affect the success of our attack, so we do not discuss it further.

7.2 Vulnerability of the PBF

The PBF has the characteristics defined in § 5.2. As with SuRF, C3(1) holds trivially. C3(2) holds because the PBF's FPR is based on its Bloom filter's FPR.

The PBF has an important property: it not only has the usual Bloom filter false positives caused by hash collisions but also has what we call *prefix false positives*. These occur when a PBF point query falsely returns positive for an input κ that is an l-bit prefix of a dataset key, simply because the Bloom filter stores both dataset keys and their l-bit prefixes. This property implies that C1 holds: with probability $1 - FPR$, an l-bit false-positive is actually the prefix of some stored key.

To show that C2 holds, we need only describe how to find prefix false positives (§ 7.2.1). Finding them makes the *IdPrefix* algorithm of C2 trivial: given an l-bit false positive κ, it outputs κ.

7.2.1 Finding l-bit false-positive keys (FindFPK)

The *FindFPK* algorithm first determines the length of key prefixes stored in the PBF, l, and then proceeds to guess prefix false positives. Crucially, finding l needs to be performed only once per attack. That is, when running the attack's rounds concurrently (§ 9), we run this step only once.

Once l is known, generating queries for uniformly random l-bit strings will find false-positive keys, similarly to the SuRF attack's *FindFPK* (§ 6.2.1). Given a set of false positive l-bit keys thus found, an expected fraction of $p/2^l$ will be prefix false positives, where p is the number of distinct l-bit prefixes of dataset keys. The remaining false positives will be hash-collision Bloom filter false positives. Because we cannot distinguish between the two types of false positives, the attack's later steps must try to extend all of them to full keys.

The crux of the *FindFPK* algorithm is to identify l. To this end, we rely on the PBF property that made it vulnerable in the first place. For any prefix length $l' \neq l$, the probability of an l'-bit key being a false positive is exactly the filter's FPR. Only for l-bit keys will we observe a "bump" in the probability of a random l-bit key being a false positive, due to the presence of prefix false positives.

Accordingly, the *FindFPK* algorithm first generates j queries for uniformly random keys of length l', for every non-trivial prefix length l' (e.g., $l' \geq 3$). It observes the fraction of false positives found and deduces that l is the length l' for which the fraction of false positives found is maximal.

8 Complexity analysis

The key factor determining prefix siphoning's effectiveness is the probability of *FindFPK* (step ① in § 5.3.2) guessing an *exploitable* key k, which is a false positive whose longest

shared prefix with stored keys is of length l, where l is a predetermined constant for which extending k into a full key is feasible (step ③).

The full version of this paper [39] includes a theoretical analysis of the SuRF and PBF attacks, which is omitted here due to space constraints. We analyze the case of uniformly random keys, which is the worst case for our attack. (If the key distribution is skewed, then (1) the guessing and full-key extraction steps can incorporate this knowledge; and (2) the prefixes SuRF stores are longer, so our attack will identify longer prefixes and thus extend them to full keys faster.)

The analysis derives the probability of *FindFPK* guessing an exploitable key. This determines the expected number of queries to guess an exploitable key or, equivalently, the number of keys we ultimately expect to extract after investing G guesses in *FindFPK*. These values also allow comparing the cost (in queries) of prefix siphoning to brute force search.

Under the realistic constraint that $|D| \ll 2^l$, where D is the dataset (e.g., $|D| = 500\,\text{M}$ and $l = 40$), we find that (1) prefix siphoning becomes more effective with growth in dataset size and better FPR—i.e., as the LSM-tree becomes more effective, so does prefix siphoning; and (2) prefix siphoning takes several orders of magnitude fewer queries to extract a key than an exhaustive brute force search.

9 Implementation issues

In previous sections, we assume the attacker can check if a key is a filter positive or negative key, based on measuring query response times. Here, we describe our implementation of this check.

The basic idea is simple. Prefix siphoning's preliminary phase (§ 5.3.1) derives a response time cutoff. Keys whose query response time is below this cutoff are considered negative; otherwise, they are considered positive. However, this cutoff only distinguishes between queries satisfied from memory and those involving I/Os. Once a query for a false-positive key completes, the I/O it performs reads the relevant SSTable into the in-memory page cache. Future queries for false-positive keys covered by this SSTable will thus get satisfied from memory.

To overcome this problem, we exploit the fact that the attack targets some production system, which is assumed to sustain heavy I/O load due its legitimate operation. This property implies that if the attacker *waits* after performing a false positive query, the SSTable brought in will be evicted from the page cache due to legitimate I/O traffic.

Unfortunately, waiting for even a few seconds after every query would make the attack impractical. We solve this challenge by performing attack rounds in a *concurrent, breadth-first* manner, as described below, instead of working depth-first (finding a false-positive key and proceeding to identify its prefix and then to extract the full key).

Step ① of § 5.3.2 (*FindFPK* execution) generates N random keys (false positive candidates) and measures a four-query average response time for each key to identify false-positive keys. The averages are computed in a breadth-first manner: there are four iterations, each of which performs one query for each key. Waiting for page cache evictions is done only between each iteration.

Step ② (*IdPrefix*) similarly executes iteratively, interleaving the next step of *IdPrefix* for each false-positive key in each iteration, until all invocations output a prefix. Again, waiting for page cache evictions is only done between iterations.

Step ③ (key extraction) likewise interleaves the searches extending each prefix. We optimize step ③'s general-case brute force suffix search by leveraging the fact that step ② outputs a set of prefixes. This enables us to discard short prefixes, so that step ③ only attempts to extend prefixes where the suffix search is feasible.

The interleaved execution of each step can be sped up using multi-core parallelization by assigning each core a subset of the N random keys, false-positive keys, or prefixes when executing step ①, ②, and ③, respectively, in the above described manner. This results in linear speedup (in the number of cores) of step execution time. Our implementation parallelizes step ③, whose execution time dominates the attack (§ 10.2.2), over 16 cores and leaves the other steps single-threaded.

10 Evaluation

In this section, we evaluate prefix siphoning attacks on SuRF and PBF in RocksDB. We demonstrate the attack's feasibility, successfully mounting it against a full-fledged RocksDB key-value store employing SuRF (§ 10.2).[3] We empirically analyze the SuRF attack's efficiency and sensitivity to data store size and filter FPR (§ 10.3). Consistent with our theoretical analysis, we find that the attack becomes more effective with growth in dataset size and better FPR—i.e., as the LSM-tree becomes more effective, so does prefix siphoning. Finally, we demonstrate the attack against the PBF (§ 10.4).

10.1 Experimental setup

Both clients and the attacked key-value store run on the same server. However, the time differences we exploit can be measured over the network using prior techniques (see § 4).

We use a server with two Intel Xeon Gold 6132 v6 (Skylake) processors, each of which has 14 2.6 GHz cores with two hyperthreads per core. The server is equipped with 192 GB DDR4 DRAM and two 0.5 TB NVMe SSDs. The server runs Ubuntu 18.04 and code is compiled with GCC 4.8.

RocksDB setup We use a version of RocksDB modified by the SuRF authors to employ SuRF [65]. The target RocksDB instance uses the NVMe devices as secondary storage. We use Linux `cgroup` to limit RocksDB's available DRAM to

[3] We use the SuRF's authors' implementation, https://github.com/efficient/SuRF.

2 GB. This configuration emulates an industrial-scale, I/O heavy key-value store setup, in which storage capacity far exceeds DRAM capacity.

The RocksDB engine stores 64-bit keys and 1000-byte values and the SuRF-Real variant. Unless noted otherwise, we use a datastore of 50 M uniformly random keys (generated using SHA1). We invoke RocksDB LSM-tree compaction after populating the datastore. We do this to emulate the compaction that naturally occurs in a real workload due to insertions, because our experiments perform only *get()*s.

Background load In all experiments, we emulate a realistic, loaded system by running 32 threads that constantly perform *get()* queries for random keys, with 50% of the queries targeting stored keys and 50% targeting non-present keys.

10.2 RocksDB+SuRF-Real key extraction

We implement the attack as described in § 9. § 10.2.1 evaluates the attack's first phase (§ 5.3.1), demonstrating that query response times can be used to distinguish negative from positive keys in practice, even in the presence of heavy background load. § 10.2.2 evaluates the attack's second phase, which extracts full keys, and compares it to a brute force search.

10.2.1 Negative/positive query time differences

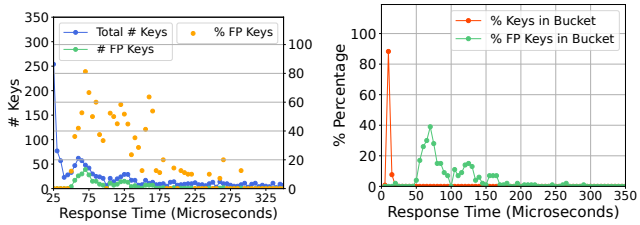
In this phase, the attacker performs 10 M *get()* queries for randomly generated keys to build the response time distribution. Table 1 shows the distribution of response times in terms of 5 microsecond buckets. The distribution is extremely skewed toward values $< 25\,\mu$s, which our attack therefore assumes are associated with negative keys.

To validate this assumption, Figure 2 visualizes the distribution while breaking the response times by queried key type (negative or false-positive). This breakdown is presented for analysis purposes; it is not available to the attacker. For readability, we present the breakdown in two ways. Figure 2(a) shows only the buckets $\geq 25\,\mu$s, which are otherwise dwarfed by the lower end of the distribution. We show both the number of keys (blue) and false-positive (green) keys in each bucket, and the percent of false-positive keys in each bucket (orange). Figure 2(b) shows the entire distribution, but bucket sizes (Y axis) are percentages instead of absolutes. For each bucket, we report the number of keys in the bucket as well as the percentage of false positives (out of all positives).

Figure 2(a) shows that the vast majority of false positive queries have a response time of 25–35 μs. Conversely, Figure 2(b) shows that this response time range contains over 50% of the false-positive keys. Overall, these results show that picking a cutoff point of 25 μs for distinguishing a negative from positive key—which is done based only on the distribution's shape, without knowledge of key types—yields a good distinguisher.

Bucket range (microseconds)	% of responses
< 5	0.77%
5 - 10	**88.3%**
$10 - 15$	7.65%
$15 - 20$	0.53%
$20 - 25$	0.05%
≥ 25	2.7%

Table 1: Distribution of query response times.



(a) Buckets $\geq 25\,\mu$s: Absolute number of queried keys

(b) All buckets: Percentage of queried keys.

Figure 2: Breakdown of query response time distribution.

10.2.2 Key extraction

The attack executes as described in § 9; specifically, wait is set to 20 seconds and each step is executed in a parallel, breadth-first manner, to minimize the amount of time spent waiting for page cache evictions. The attacker generates a set of 10 M random keys to find false-positive keys (step ① of § 5.3.2). The attacker next identifies the prefix each false-positive key shares with a stored key (step ②). Finally, the attacker discards every prefix of length < 40 bits and attempts to extend the remaining prefixes into full keys (step ③).

Figure 3 shows the number of keys extracted as a function of the number of total number of *get()* requests issued by the attack (aggregated over steps ①–③). The figure also compares the attack to an *idealized* attack, which uses internal RocksDB debugging counters to accurately determine the filters' responses for each queried key, instead of relying on query response times.

Because the idealized attack never incorrectly classifies a key, it identifies more false positives than the actual attack in step ①. It thus requires more queries in step ② to identify the shared prefixes of the keys provided to step ②, as there are more of them. Consequently, the idealized attack begins step ③ later (in terms of queries) than the actual attack, which is why its line is "shifted" compared to actual attack. For this reason, the idealized attack also requires more queries overall. Ultimately, however, the actual attack extracts only 74 fewer keys than the idealized version.

The idealized attack is also faster (in real time) than the actual attack, because it does not require waiting for page cache evictions. The actual attack's key extraction rate is $\approx$ 10 minutes/key, while the idealized attack achieves 0.2 minutes/key.

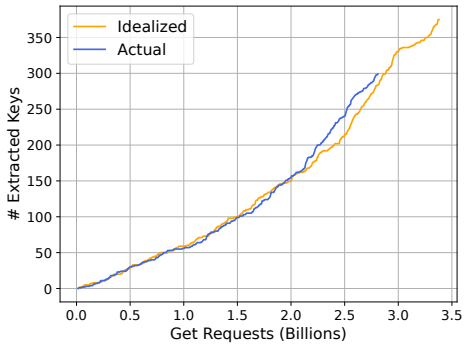


Figure 3: Actual vs. idealized prefix siphoning against SuRF-Real: Number of keys extracted as attack progresses.

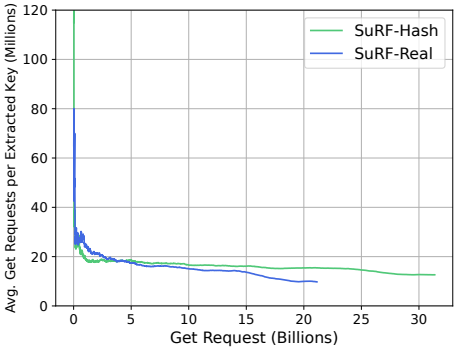


Figure 4: SuRF-Hash vs. SuRF-Real: Moving average of queries per extracted key as a function of attack progress (measured in queries).

attack step	# queries (millions)	queries/total (%)
① Find false positives	10M	0.35%
② Identify prefixes	0.025M	0.0009%
③ Extract keys	2581M	91.68%
Wasted	224M	7.9%

Table 2: Attack queries per stage. Wasted queries futilely attempt to extend an incorrectly identified prefix into a full key.

Table 2 shows a breakdown of the (actual) attack's queries across all three steps. The bulk of the attack is spent on step ③, extending prefixes into full keys. Our later analysis (§ 10.3.2) explains this number. The table also reports wasted queries, which are issued when the attack futilely tries to extract a key from an incorrect prefix, which was misidentified due to incorrectly classifying a key as a false-positive (based on its query response time). Additional wasted queries (not shown) are spent identifying prefixes of length < 40 bits in steps ①–②, which are then discarded. While over 90% of prefixes identified by steps ①–② are discarded, this waste is negligible, as they are discarded before the most expensive step.

Comparison to brute force We further evaluate a brute force attack, that randomly guesses keys until a stored key is found. We allow this attack to run for 10× more time than the prefix siphoning experiment—but it fails to guess a key. Unsurprisingly, brute force search for a large key space is infeasible.

SuRF-Hash vs. SuRF-Real SuRF-Hash complicates the attack. Compared to SuRF-Real with the same per-key space budget, SuRF-Hash replaces key bits (SuRF-Real's suffix bits) with hash value bits. This means that possible prefixes to identify are shorter and that the filter's FPR is lower, making the number of false positives identified in step ② lower. On the other hand, as discussed in § 6.2.2, when identifying the prefixes and performing key extraction, the attacker can use the false-positive key's hash value to ignore definitely incorrect guess—potentially improving the attack's efficiency.

To evaluate this trade-off, we compare idealized attacks against the same dataset, with RocksDB using either SuRF-Real with 8-bit suffixes or SuRF-Hash with 8-bit hashes. Thus, in SuRF-Hash, the suffix search space when extracting a key 256× larger than in SuRF-Real, but the attacker will ignore $255/256$ of its guesses on average. To compensate for SuRF-Hash's lower FPR, the initial false-positive key search of the SuRF-Hash attack uses 3× the number of candidate keys used for SuRF-Real. Figure 4 therefore compares the attacks' amortized cost, in terms of a moving average of queries per extracted key as a function of attack progress.[4] The SuRF-Hash attack's extra initial queries (for finding false positives) manifest as the peak of the per-key cost, when all these extra queries are amortized across only a handful of keys. The extra cost is eventually amortized away, into a per-key cost of 12 M vs. 10 M queries for SuRF-Hash vs. SuRF-Real, respectively. For this similar cost, the SuRF-Hash attack extracts 2490 keys vs. 2171 keys for the SuRF-Real attack.

10.3 Attack analysis

This section analyzes the attack's efficiency (§ 10.3.1) and sensitivity to data store size (§ 10.3.2) and filter FPR (§ 10.3.3).

10.3.1 Efficiency

Figure 5 shows the attack's efficiency, measured as average *get()*s per extracted key as a function of attack progress. We compare across three 50 M random 64-bit key sets to show the results are not a function of the specific key set.

The average number of queries per extracted key converges to about 9 M $\approx 2^{23}$. This indicates that the attack extracts keys with roughly the work required to search a 23-bit space—40992× better than a brute force search of the full key space ($2^{64}/50\,\text{M} \approx 2^{38.4}$). The attack also extracts a substantial number of keys (375, 419, and 423 keys).

[4] I.e., the Y axis reports the number of *get()*s issued divided by the number of keys extracted up to the current X-axis point.

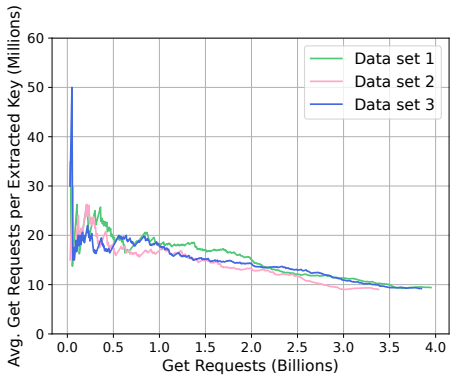


Figure 5: Attack efficiency: average number of *get()*s per extracted key as attack progresses.

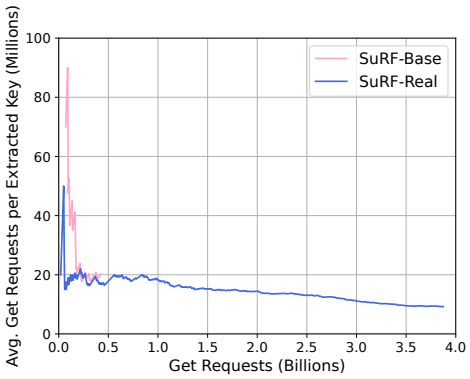


Figure 7: SuRF-Real vs. SuRF-Base: Moving average of queries per extracted key as a function of attack progress (measured in queries).

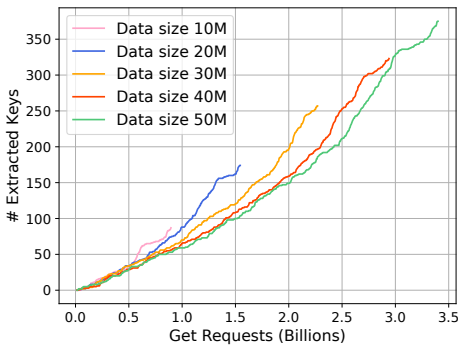


Figure 6: Idealized attack against SuRF-Real: Number of keys extracted for different dataset sizes.

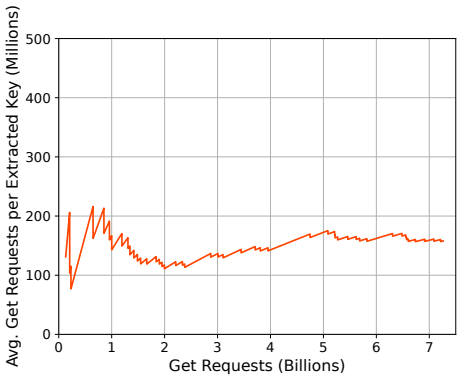


Figure 8: Idealized prefix siphoning against PBF ($l = 40$ bits).

10.3.2 Sensitivity to dataset size

To evaluate the attack's sensitivity to the dataset size, we progressively shrink our original 50 M key set into smaller subsets of size $c \cdot 10$ M keys for $c \in [1,5]$. We then perform an idealized attack against the system with each dataset, but using the same set of random keys for step ①, so any difference in attack behavior can related only to the datastore size and not the key distribution.

Figure 6 shows the number of keys extracted as the attack progresses. Prefix siphoning is more effective as the dataset size increases: it extracts ≈ 100 keys from the 10 M dataset, but almost 400 keys from the 50 M dataset.

10.3.3 Sensitivity to SuRF FPR

We show that prefix siphoning becomes more effective as SuRF's FPR improves, i.e., the attack becomes more harmful to the system as SuRF becomes more productive to the system. To demonstrate this effect, we compare idealized attacks against the same dataset, with RocksDB using either SuRF-Base or SuRF-Real. SuRF-Base stores shared key prefixes, padded to the next full byte (which adds 1–8 bits to the prefix). SuRF-Real does the same, plus stores a byte from the key's unique suffix, and thereby improves its FPR (see § 6.1).

We carry out the attacks against each SuRF variant using the same initial random key set, used to identify false-positive keys. Figure 7 reports the attack's amortized cost (queries per extracted key) as the attack progresses.

In both cases, the attack has similar efficiency of ≈ 10 M queries per extracted key, as evident from the similar slope of the two lines. However, the attack is more successful against SuRF-Real, where it extracts 420 keys, than against SuRF-Base, where is extracts 21 keys. The reason for the improved effectiveness is that SuRF-Real's extra key byte storage makes an initial false-positive key much more likely to have a prefix length of > 40 bits, resulting in more false positives making it to step ③.

The situation is similar with SuRF-Hash, which further improves the FPR over SuRF-Real (Figure 4). As mentioned in § 10.2.2, the idealized SuRF-Hash attack extracts 2490 keys vs. 2171 keys for the idealized SuRF-Real attack.

10.4 RocksDB+PBF key extraction

We evaluate an idealized prefix siphoning attack against RocksDB's PBF. We use a dataset of 50 M uniformly random 64-bit keys. We configure the PBF to store prefixes of length

$l = 40$ bits and to consume 18 bits/key (which is roughly the space usage of SuRF in our experiments).

Step ① (*FindFPK*) perform 1 M queries for uniformly random 40-bit keys, which result in 457 false-positive keys. The attack then attempts to extend these false positives into full keys. It eventually extracts 46 keys, which matches the expected number of prefix false positives observed in 1 M random guesses ($1\text{M} \cdot 50\,\text{M}/2^{40} = 45.4$). Figure 8 plots the attack's amortized cost (queries per extracted key) as the attack progresses. The PBF attack makes 160 M queries per extracted key, which is 20× more queries/key than the SuRF attack, but still three orders of magnitude better than a brute force search. The reason for this difference is that the PBF attack wastes effort trying to extend Bloom filter false positives that are not prefix false positives.

11 Mitigation

Here, we discuss approaches for mitigating prefix siphoning attacks. Unfortunately, every potential solution constitutes some trade-off, whether in query performance, memory efficiency, complexity, or other system aspects.

System-level approaches A system can block prefix siphoning attacks by only querying its key-value storage engine for keys the requesting user is allowed to access. This approach requires re-architecting the system so that a key's ACL is kept outside of the key-value store. In addition, a system can rate limit user requests, thereby slowing down prefix siphoning attacks. This approach is viable only if the system is not meant to handle a high rate of normal, benign requests.

Key-value store mitigation A key-value engine can block prefix siphoning by maintaining separate filters for point and range queries for each SSTable file. Unfortunately, this approach will double filter memory consumption. In addition, it will not block attacks that target range queries (which we believe are possible, and are currently exploring).

Filter-level mitigation A natural mitigation is for key-value stores to employ non-vulnerable range filters. Like the separate filter approach described above, this mitigation carries the risk of being vulnerable to future extensions of prefix siphoning to range queries.

In addition, the properties that make a range filter non-vulnerable to point query-based prefix siphoning may limit its utility in practice. For example, Rosetta (Robust Space-Time Optimized Range Filter) [45] is a range filter that does not conform to our vulnerable range filter characterization (§ 5.2), but it lacks support for variable-length keys, which are important in practice.

Rosetta uses Bloom filters for SuRF-like prefix-based filtering. Rosetta assumes a bound on the possible key length in bits, L. A Rosetta instance consists of L Bloom filters, $B_1, \ldots, B_L$. When a key k is inserted into the filter, each i-bit prefix is inserted into the i-th Bloom filter B_i. A Rosetta point query thus simply queries B_L, making Rosetta non-vulnerable to prefix siphoning.

The Rosetta paper does not specify how variable-length keys are handled. Its design is clearly incompatible with such keys if there is no predetermined bound on their size. Even if such a bound exists (and can thus be used for L), Rosetta requires every key to be padded to L bits, so that point queries function correctly. This requirement significantly increases the filter's memory consumption.

Encrypted key-value stores Disclosed keys reveal no sensitive information if they are stored encrypted in the storage engine. However, encrypting key-value pairs requires re-architecting the entire system so it can query on encrypted data [63, 64]. Most if not all deployed key-value stores do not support such encryption.

12 Related Work

Key-value store timing attacks Existing key-value store timing attacks aim to disclose stored values. These attacks work by exploiting external mechanisms such as memory deduplication [55] or memory compression [56], which can be disabled for protection. In contrast, prefix siphoning exploits a mechanism of the key-value store itself, which cannot be disabled for protection without suffering significant throughput degradation and additional I/O traffic.

Storage engine timing attacks Timing attacks mostly target cryptographic software rather than storage engines. Futoransky et al. [35] extract private keys from a MySQL database with a timing attack, but the attack relies on insertions of attacker-chosen data. Wang et al. [59] show a practical timing attack on a multi-user search system, such as Elasticsearch.

13 Conclusion

This paper shows that certain range filters make LSM-trees vulnerable to novel *prefix siphoning* timing attacks, which exploit differences in query response times to reveal keys and prefixes of keys stored in the LSM-tree. Our results show that key-value store performance improvements may trade security in exchange, and encourage practitioners and researchers to evaluate the security impact of their work. We also hope that our characterization of vulnerable range filters will spur research on more secure filters.

Acknowledgments

We extend our deepest thanks to Yuvraj Patel, the paper's shepherd, and the anonymous reviewers for their dedication and assistance in improving this paper and their valuable feedback. We thank Guy Khazma for his work on an earlier stage of this project.

References

[1] CEPH. https://github.com/ceph/ceph.

[2] MongoDB. https://www.mongodb.com/.

[3] MySQL Server. https://github.com/mysql/mysql-server.

[4] Amazon. Amazon S3. https://aws.amazon.com/s3/, 2020.

[5] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, 2009.

[6] Apache. Apache Flink — Stateful Computations over Data Streams. https://flink.apache.org, 2022.

[7] Mikhail Bautin. How We Built a High Performance Document Store on RocksDB? https://www.yugabyte.com/blog/how-we-built-a-high-performance-document-store-on-rocksdb/, 2019.

[8] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't Thrash: How to Cache Your Hash on Flash. In *VLDB*, 2012.

[9] Peter Benjamin. s3-fuzzer. https://github.com/pbnj/s3-fuzzer, 2017.

[10] Burton H. Bloom. Space / Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7), 1970.

[11] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. In *USENIX Security Symposium*, 2003.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *FAST*, 2020.

[13] 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. *ACM TOCS*, 26(2), 2008.

[14] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime Data Processing at Facebook. In *SIGMOD*, 2016.

[15] Alex Conway, Martín Farach-Colton, and Philip Shilane. Optimal Hashing in External Memory. In *ICALP*, 2018.

[16] 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 *USENIX ATC*, 2020.

[17] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and Limits of Remote Timing Attacks. *ACM Transactions on Information and System Security*, 12(3), 2009.

[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD*, 2017.

[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM TODS*, 43(4), 2018.

[20] Niv Dayan and Stratos Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *SIGMOD*, 2018.

[21] Niv Dayan and Stratos Idreos. The Log-Structured Merge-Bush & the Wacky Continuum. In *SIGMOD*, 2019.

[22] Niv Dayan and Moshe Twitto. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD*, 2021.

[23] Tom de Vries. Teh s3 bucketeers. https://github.com/tomdev/teh_s3_bucketeers/, 2021.

[24] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-Based Storage. In *SIGMOD*, 2011.

[25] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Stumm. Optimizing Space Amplification in RocksDB. In *CIDR*, 2017.

[26] Phil Eaton. What's the big deal about key-value databases like FoundationDB and RocksDB? https://notes.eatonphil.com/whats-the-big-deal-about-key-value-databases.html, 2022.

[27] Nathan Eddy. Cloud Misconfig Exposes 3TB of Sensitive Airport Data in Amazon S3 Bucket: 'Lives at Stake'. https://www.darkreading.com/application-security/cloud-misconfig-exposes-3tb-sensitive-airport-data-amazon-s3-bucket, 2022.

[28] David Eisenstat. Structured data encoding in CockroachDB SQL. https://github.com/cockroachdb/

cockroach/blob/master/docs/tech-notes/encoding.md, 2021.

[29] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In *SIGCOMM*, 2012.

[30] Facebook. RocksDB. https://github.com/facebook/rocksdb.

[31] Facebook. MyRocks. http://myrocks.io/, 2015.

[32] Facebook. MyRocks record format. https://github.com/facebook/mysql-5.6/wiki/MyRocks-record-format, 2019.

[33] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *CoNEXT*, 2014.

[34] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[35] Ariel Futoransky, Damián Saura, and Ariel Waissbein. The ND2DB Attack: Database Content Extraction Using Timing Attacks on the Indexing Algorithms. In *WOOT*, 2007.

[36] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security Symposium*, 2020.

[37] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-Structured Data Stores. In *EuroSys*, 2015.

[38] Google. LevelDB. https://github.com/google/leveldb.

[39] Adi Kaufman, Moshik Hershcovitch, and Adam Morrison. Prefix Siphoning: Exploiting LSM-Tree Range Filters For Information Disclosure (Full Version). *arXiv e-prints*, abs/2306.04602, 2023.

[40] Redis Lab. Redis. https://github.com/redis/redis.

[41] Cockroach Labs. CockroachDB. https://www.cockroachlabs.com/, 2022.

[42] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2), 2010.

[43] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP*, 2011.

[44] LinkedIn. FollowFeed: LinkedIn's Feed Made Faster and Smarter. https://engineering.linkedin.com/blog/2016/03/followfeed--linkedin-s-feed-made-faster-and-smarter, 2016.

[45] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *SIGMOD*, 2020.

[46] Netflix. Application Data Caching using SSDs. http://techblog.netflix.com/2016/05/application-data-caching-using-ssds.html, 2016.

[47] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[48] OWASP. Insecure Direct Object Reference Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html, 2021.

[49] 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.

[50] Jordan Potti. Awsbucketdump. https://github.com/jordanpotti/AWSBucketDump, 2018.

[51] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *SOSP*, 2017.

[52] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. LittleTable: A Time-Series Database and Its Uses. In *SIGMOD*, 2017.

[53] Dan Salmon. S3scanner. https://github.com/sa7mon/S3Scanner, 2022.

[54] Apache Samza. State Management. http://samza.apache.org/learn/documentation/0.8/container/state-management.html, 2017.

[55] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote Memory-Deduplication Attacks. In *NDSS*, 2022.

[56] Martin Schwarzl, Pietro Borrello Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical Timing Side-Channel Attacks on Memory Compression. In *IEEE S&P*, 2023.

[57] Russell Sears and Raghu Ramakrishnan. BLSM: A General Purpose Log Structured Merge Tree. In *SIGMOD*, 2012.

[58] Mark Sutherland, Babak Falsafi, and Alexandros Daglis. Cooperative Concurrency Control for Write-Intensive Key-Value Workloads. In *ASPLOS*, 2023.

[59] Liang Wang, Paul Grubbs, Jiahui Lu, Vincent Bindschaedler, David Cash, and Thomas Ristenpart. Side-Channel Attacks on Shared Search Indexes. In *IEEE S&P*, 2017.

[60] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *EuroSys*, 2014.

[61] Brian Warehime. insp3ctor. https://github.com/brianwarehime/inSp3ctor, 2018.

[62] Ian Williams. Bucket finder. https://github.com/FishermansEnemy/bucket_finder, 2013.

[63] Xingliang Yuan, Yu Guo, Xinyu Wang, Cong Wang, Baochun Li, and Xiaohua Jia. EncKV: An Encrypted Key-Value Store with Rich Queries. In *ASIA CCS*, 2017.

[64] Xingliang Yuan, Xinyu Wang, Cong Wang, Chen Qian, and Jianxiong Lin. Building an Encrypted, Distributed, and Searchable Key-Value Store. In *ASIA CCS*, 2016.

[65] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD*, 2018.

EPF: Evil Packet Filter

Di Jin
Brown University

Vaggelis Atlidakis
Brown University

Vasileios P. Kemerlis
Brown University

Abstract

The OS kernel is at the forefront of a system's security. Therefore, its own security is crucial for the correctness and integrity of user applications. With a plethora of bugs continuously discovered in OS kernel code, defenses and mitigations are essential for practical kernel security. One important defense strategy is to isolate user-controlled memory from kernel-accessible memory, in order to mitigate attacks like ret2usr and ret2dir. We present EPF (Evil Packet Filter): a new method for bypassing various (both deployed and proposed) kernel isolation techniques by abusing the BPF infrastructure of the Linux kernel: i.e., by leveraging BPF code, provided by unprivileged users/programs, as attack payloads. We demonstrate two different EPF instances, namely BPF-Reuse and BPF-ROP, which utilize malicious BPF payloads to mount privilege escalation attacks in both 32- and 64-bit x86 platforms. We also present the design, implementation, and evaluation of a set of defenses to enforce the isolation between BPF instructions and benign kernel data, and the integrity of BPF program execution, effectively providing protection against EPF-based attacks. Our implemented defenses show minimal overhead ($< 3\%$) in BPF-heavy tasks.

1 Introduction

The security of a computer system can only be as good as that of the underlying OS kernel. The kernel provides a relatively simplistic abstraction for programs to build on top of, and mediates their access to system resources. Hence, the confidentiality and integrity of user programs rely solely on the security of the OS kernel itself. Yet, the kernel is hard to defend, due to its unique execution model, and the sheer size and complexity of its code. With the development of automated kernel-code testing tools, such as syzkaller [13], thousands of bugs have been found across different OSes [12]. It is even pointed out that bugs are discovered faster than they are fixed [113]. The abundance of errors and vulnerabilities in OSes amplifies the importance of protection mechanisms that reduce the exploitation potential of kernel vulnerabilities.

Standard defenses, such as W^X [6] and ASLR [6], are also adopted by the Linux kernel, aiming to stop and mitigate code-injection- [64] and code-reuse-based [40] attacks. More specifically, these defenses aim to limit the attacker's ability to successfully mount an attack. However, attacks such as ret2usr [68], which completely encode their payloads in userspace, indicate that kernel exploitation is made significantly easier by the relatively *weak separation* between the kernel and userspace (as opposed to a user application, such as web server, where the interface between it and the clients is much more well-defined and limited). Similarly, protection mechanisms like SMEP [112] try to stop kernel attacks by limiting the attacker's ability to encode attack payloads in a kernel-accessible manner (more attacks and defenses in this direction are discussed in Section 2.1).

In this paper, we explore the possibility of abusing the *BPF infrastructure* in the Linux kernel—more specifically, by leveraging BPF programs as attack payloads. We have identified three properties that make BPF programs a promising candidate for such a task. First, BPF programs can be created by *unprivileged users*, and contain memory contents chosen by the user (such that they can be used to encode malicious contents). Second, BPF programs can be created in *large amounts*, such that during exploitation valid references to them can be constructed (e.g., by just guessing) with good probability. Third, BPF programs are created by users, but "*consumed*" by the OS kernel. Access to such payloads cannot be prevented by existing, strong kernel-user isolation mechanisms (e.g., XPFO [67]), because they cannot be differentiated from regular data that the kernel operates upon.

Yet, unlike previous techniques where the payload content is encoded inside regions acting effectively as "byte buffers", a BPF programs's in-memory representation has a non-trivial structure. To address this problem, we develop special code-reuse strategies, dubbed as *EPF*, to utilize payloads with the constraints introduced by the corresponding BPF structure. BPF programs can aid exploitation because they are not strongly-isolated from normal kernel data.

And to defend against such EPF-style attacks, we develop a set of defenses that enforce strong BPF-to-kernel isolation. Recognizing the fact that BPF is essentially a "virtual architecture", we follow a roadmap similar to that of isolating native code from pure data: (1) BPF code should not be read or written as regular, kernel data; (2) regular kernel data should not be executed as BPF code; and (3) BPF code should not be reused as semantically different BPF code.

To summarize, we make the following contributions:

- We introduce a novel, high-volume, undefended method to inject payloads in kernel space for aiding the exploitation of memory errors in kernel code, and we create systematic techniques to utilize them in different architectures. Our methods enable attacks that bypass state-of-the-art defenses that focus on enhancing kernel-user isolation.
- We develop defenses against exploitation that (ab)uses the BPF infrastructure by enforcing strong isolation between BPF code and regular kernel data, and the integrity of BPF execution under attack.
- We evaluate both our attacks and defenses. We create exploits using our techniques on four different real-world vulnerabilities. We integrate our defenses into Linux kernel and demonstrate low overhead ($< 3\%$) on BPF-heavy tasks.

2 Background

2.1 Kernel Exploitation and Defense

Exploitation is the process of tampering with a victim system/-software codebase by abusing vulnerabilities in it [95]. In the case of operating systems (OSes), because OS kernels are typically written in memory- and type-unsafe languages, like C, C++, and ASM, the most common approach to their exploitation entails abusing memory errors in kernel code [73,74,88,96]. In general, two are the dominant exploitation strategies (re: memory errors): *code-injection* [64] and *code-reuse* [40]. Code-injection leverages memory corruption vulnerabilities to place malicious code (i.e., *shellcode*) in the victim's address space before corrupting control data (e.g., *return address*, *function pointers*, *dispatch tables*) to steer execution to it. In contrast, code-reuse stitches together *existing* (i.e., benign) code snippets, in an out-of-context manner, to perform the respective (malicious) computation.

In addition to the above, there exist *kernel-specific* exploitation techniques that address unique challenges of the OS kernel setting. Throughout the evolution of kernel attacks and defenses, regarding memory errors, the ability to create attacker-controlled, exploit-time-accessible payloads has been at the forefront of the subject matter. For example, in *ret2usr* [68] attacks, the adversary first corrupts a code pointer, and then diverts the control flow to userspace code, tricking the kernel into executing malicious (shell)code with elevated privileges. Alternatively, a different flavor of such an attack employs code-reuse techniques (e.g., ROP [100]) with payloads placed in userspace [74, 77]. An essential property of ret2usr-like attacks is the placement of their payload (e.g., shellcode or code-reuse payload) in userspace.

To provide protection against exploits that follow the ret2usr approach, various defenses focus on stopping the respective payloads from being *accessible*. CPU features such as SMEP [112], SMAP [49], PXN [2], PAN [7], as well as software solutions such as kGuard [68], PaX's KERNEXEC [89] and UDEREF [90, 91], were introduced to prevent userland code/data from being executed/accessed freely by the OS kernel. Seeking new ways to provide payloads for exploiting kernel vulnerabilities, without accessing userspace, in the past we proposed to (ab)use the implicit memory sharing between userspace and the kernel: i.e., the `physmap` region [67].

For performance reasons, modern OS kernels keep a continuous mapping of the physical memory (or part of it) in kernel space, which naturally contains user-controlled content. In such attacks, dubbed *ret2dir*, the adversary tries to allocate enough physical pages containing the respective payload, and through the implicit sharing of `physmap`, the payload will be utilized/accessed in a later stage via code injection or reuse. To defend against ret2dir-based attacks, we introduced the concept of *XPFO* (eXclusive Page Frame Ownership) [67]. XPFO prevents the kernel from accessing any memory page that houses userland content, using a kernel (`physmap`-resident) address. Again, the defense impedes the attackers' ability to access their payload.

In this work, we show that unintended access and implicit sharing are not the only reliable sources of *payload injection*. As it turns out, the ability to "push" BPF programs [83] in kernel space provides the attacker with enough control over the contents of kernel memory, to the extent that BPF can be used as an *arbitrary* payload-encoding mechanism. As BPF programs are designed to *live in*, and *used by*, the OS kernel, such payloads bypass all defenses that rely on the strong isolation of kernel- from user-space [66].

2.2 BSD Packet Filter

Design and Usage The BSD Packet Filter (BPF) [83] was originally designed for *filtering* packets during network monitoring: by providing the kernel with "instructions" regarding how to filter packets, before delivering them to a monitoring process in userspace, BPF eliminates unnecessary data copying and context switching. The design and implementation of the Linux Socket Filter (LSF; i.e., our main subject of study) [60] was inspired heavily by BPF. However, it has recently evolved into a generic utility, acting as a *universal* in-kernel virtual machine [46]: its execution is strictly *sandboxed*, and no unintended *side-effects* escape confinement.

A wide range of kernel components, and applications, make use of the expressiveness and security provided by BPF. Docker [10], Firefox [9], and Chromium [1], as well as automated system call (syscall) filtering schemes, like `sysfilter` [52], Confine [55], and Chestnut [42], use `seccomp-BPF` [80] to specify syscall filtering policies. In addition, BPF programs can be attached to Kprobes (kernel probes) [105], giving rise to powerful kernel tracing tools [38], while BPF-based networking frameworks are developed to enable agile packet processing [15, 78]. Lastly, BPF programs have also been proposed to be used in FUSE (Filesystem in USErspace) [34] to reduce context switching overheads.

Features Two different flavors of BPF exist in Linux: classic BPF (cBPF); and extended BPF (eBPF). Internally, cBPF is converted to eBPF, and therefore only a single execution engine (for eBPF) exists nowadays [70]. cBPF programs are used for socket filtering (`setsockopt`), and syscall filtering (`prctl`, `seccomp`), while eBPF programs are managed with the `bpf` syscall, allowing kernel subsystems to implement different methods for attaching/invoking eBPF code.

cBPF is similar to the original BPF, with two general-purpose registers and 16 (addressable) scratch memory slots, all of which are 32-bit wide. In contrast, eBPF has 10 general-purpose, 64-bit registers, and is equipped with a set of helper functions to access either eBPF *maps* (i.e., a family of key/-value store data structures) or other, *internal functionality* (like getting the current process-ID or generating random numbers) [3]. eBPF maps can be made accessible from multiple eBPF programs or user processes [4].

Both cBPF and eBPF specify a RISC-like instruction set, allowing: (a) loading and storing operations (re: immediate values or scratch memory slots); (b) moving values between registers; (c) performing arithmetic and logical operations; and (d) branching. To guarantee termination, there is no *indirect* branching, and the branch instructions can only *jump forward*. In addition, eBPF provides specific instructions for invoking helper functions and other eBPF programs. JIT (Just-In-Time) compiling for eBPF [45] instructions, down to machine code, allows for performance gains [99], compared to using the eBPF interpreter. Lastly, popular tools have also added support for eBPF. LLVM supports BPF as a backend since v3.7 [81], so that developers can choose to write BPF programs using a syntax similar to C. Similarly, BCC [61] and `libbpf` [8] are frameworks that allow developers to easily create and interact with loaded BPF programs.

Security The isolation between the BPF runtime and OS kernel is crucial for the security of the latter. For performance reasons, BPF favors static, *ahead-of-time* checking (over dynamic, runtime checking), when enforcing such isolation. The static checker for cBPF ensures the following properties: (1) jumps (i.e., branches) do not go backwards; and (2) scratch space accesses target initialized locations only. Any program that cannot be statically verified by the checker is rejected. In case of packet filtering, the validity of access to a packet cannot be determined statically, as packet sizes may differ at runtime. Hence, such accesses will be translated to calling helper functions, and bounds are enforced at runtime.

The checker for eBPF is more complex [84]. Same as the cBPF checker, it needs to make sure that control flow does not go backwards. It also validates, and replaces, eBPF map and helper function references, and analyzes register value types, such that the corresponding operations lead to predictable memory accesses to safe locations. The verifier is known to be prone to errors, such as incorrect value range analysis [16, 17, 21] and insufficient protection against speculative execution-related vulnerabilities [19, 22]. Hence, the unprivileged `bpf` syscall is disabled by default [48] (`defconfig` in x86 Linux), while distributions are adopting a "on/off" choice [51, 106].

BPF has also been used as aid in the exploitation of *transient execution* vulnerabilities [72], as well as in settings that involve limited memory corruption capabilities (wrt spatial ranges and/or value choices) [63, 107]. Concerns regarding the former have led to removing the interpreter entirely, and always using JIT-compiled BPF, in certain settings, in order to reduce the risk of Spectre-like attacks (i.e., via `CONFIG_BPF_JIT_ALWAYS_ON` [103]). However, BPF JIT comes with its own set of security concerns.

First, because BPF JIT provides fine-grain control of native code (i.e., instructions) in kernel space, it is favored by specific transient-execution attacks [32, 71] (against the OS kernel). Second, it has been shown that the JIT engine can be used to facilitate *code injection* [82, 98], and hardening techniques such as *constant blinding* [36] and *code-offset randomization* [53] are added to counter these attacks. Lastly, the JIT compiler itself also suffers from errors. Nelson et al. [86] proposed the use of formal methods to ensure the correctness of JIT compilation, in which they report 82 bugs, demonstrating the difficulty of developing a correct (BPF) JIT compiler. The above issues have made the choice between the BPF interpreter and JIT compiler a difficult one, because both sides come with different security trade-offs. Notably, `defconfig` in x86 Linux, as well as popular distributions, like Debian, choose to keep the BPF interpreter compiled-in, which is what we assume in this work.

3 Threat Model

3.1 Adversarial Capabilities

We consider an attacker who is a local, *unprivileged* Linux user, aiming at escalating their privileges. More specifically:

Unprivileged Access The attacker is able to perform anything an unprivileged user can, like executing arbitrary code in userspace, invoking syscalls, accessing the filesystem, and interacting with OS interfaces (`procfs` [14], `sysfs` [11]).

BPF Functionality The attacker has the ability to *push* cBPF programs in kernel space. First, BPF should be enabled in the kernel. This is true for any Linux kernel compiled with

network support, because the BPF subsystem is turned on by `CONFIG_BPF`, which is selected by `CONFIG_NET` [104]. Second, the attacker should have access to the BPF infrastructure, which is always the case because the `setsockopt`, `seccomp`, and `prctl` syscalls are *not* privileged. Third, the interpreter needs to exist in kernel code, which means that `CONFIG_BPF_JIT_ALWAYS_ON` should be off (§2.2). We do not require the `bpf` syscall being available to the attacker, which can be used to create eBPF programs. Actually, the *unprivileged* `bpf` syscall is disabled in all major distributions [51, 106].

Memory Errors We also assume that the attacker has access to a (at least one) memory corruption vulnerability in kernel code, and that by abusing this vulnerability they are able to overwrite code and data pointers, either in a *temporal* (e.g., use-after-free [23, 24]) or *spatial* (e.g., out-of-bound access [27, 28]) manner. We do not assume any error, or bug, in the BPF interpreter, verifier, or JIT compiler. Although the correctness of these components is challenging [25, 26, 86], this is something orthogonal to our attack(s)/EPF.

3.2 Hardening Assumptions

On the kernel side, we assume that W^X [6] is enforced, such that code injection is not possible. In addition, we consider that the kernel is hardened against ret2usr attacks, using SMEP [112]/PXN [2] and SMAP [49]/PAN [7]. Furthermore, we assume that no implicit memory sharing can take place between userland processes and the OS kernel, and hence ret2dir attacks are not attainable. Note that this is a strong assumption, as currently Linux does not employ a comprehensive defense against ret2dir, like XPFO [67]. We also assume that the page tables are protected against tampering with page table integrity mechanisms, such as PT-Rand [50] or xMP [94]. Lastly, kernel ASLR [54] is orthogonal to our attack(s); if deployed, EPF leverages known techniques to bypass it (§7).

4 Evil Packet Filter

We use the term EPF (Evil Packet Filter) to refer to a set of attacks that (ab)use the BPF infrastructure for injecting malicious payloads in kernel space. EPF allows bypassing existing isolation mechanisms [7, 49], which prevent user-controlled content from aiding kernel exploitation (§3.2). Due to the nature of the in-memory representation of BPF programs, making use of them as an attack vector is quite challenging. First, we describe certain design/implementation details of BPF, how BPF programs are created, what malicious content can be "hidden" in them, and how an attacker can locate them. Then, we describe two EPF-based attacks: *BPF-Reuse* (EPF v1—variant 1) and *BPF-ROP* (EPF v2—variant 2).

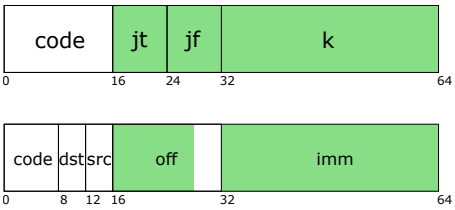


Figure 1: Fields and their sizes (in bits) in cBPF (top) and eBPF (bottom) instructions. Green regions are the parts of the instruction that can be controlled by an attacker.

4.1 Linux BPF Internals

BPF-code Management cBPF programs are primarily created via `setsockopt`, `prctl`, and `seccomp`, which can be used to "push" (cBPF) filtering code in kernel space; eBPF code can be copied in kernel space using the `bpf` syscall.

Both cBPF and eBPF programs are represented using the same data structure: that is, `struct bpf_prog`. We refer to this data structure as the "BPF program"; BPF programs pushed by `setsockopt`, `prctl`, and `seccomp` are considered 'cBPF', while those copied in kernel space by `bpf` are 'eBPF'. `struct bpf_prog` is always *page-aligned* when allocated. After a cBPF program is loaded into kernel space, the cBPF instruction array is duplicated, and stored separately—we call this instruction array as the *original cBPF code*. This code is referenced from `struct bpf_prog` by a member pointer `orig_prog`, allowing the corresponding process to retrieve the original cBPF code later (if needed). Then, the cBPF code is statically verified for safety, and translated *in-place*, becoming a *verified BPF program* (§2.2). Notably, the verification of cBPF programs is different from the one of eBPF programs, and after the cBPF instructions are translated to eBPF instructions, they are not verified again.

Lastly, a verified BPF program goes (optionally) through the process of being JIT-compiled (when `/proc/sys/net/core/bpf_jit_enable = 1`), while the memory permissions of pages that host the BPF program become *read-only*.

Data Structures The BPF program data structure (in the case of cBPF) consists of a header, which includes a pointer to the *interpreter* function, and a pointer to the original cBPF code, and an array of instructions. After verification, the array of instructions contains only eBPF code, due to the in-place cBPF⇝eBPF translation. (We refer to this array as the eBPF code. Notice that, internally, both cBPF and eBPF programs are represented with eBPF code.) BPF programs are allocated from the `vmalloc` region, whereas the original cBPF code is duplicated using `kmalloc` and lives in `physmap` [67].

In cBPF instructions (Figure 1, top), the `code` field is the opcode, defining the respective operation; `jt` and `jf` are two fields used for specifying where to jump if a predicate is true or false; and `k` is used for encoding immediates. Similarly, in eBPF instructions (Figure 1, bottom), both `src` and `dst` encode a number between 0–10, corresponding to one of the

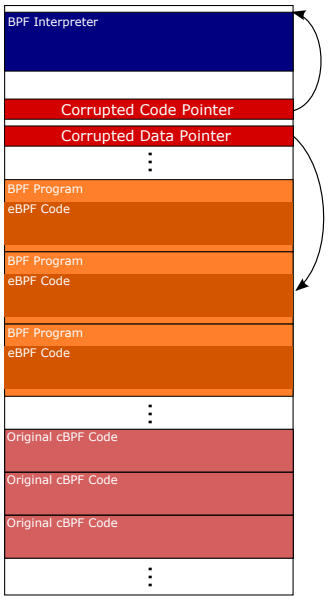


Figure 2: Memory layout during an EPF v1 (BPF-Reuse) attack. The corrupted code pointer points to the BPF interpreter function, and, at the time of its invocation, its argument points to one of the malicious BPF programs.

10 general-purpose registers or the frame pointer; the `off` field is used by memory load/store instructions, and jump instructions, to specify the offset of such operations; and `imm` stores the value of instructions that require an immediate.

Payload-encoding Challenges Ideally, the attacker would like the memory region they control to be of *arbitrary* size and content. However, this is not the case for BPF. Both cBPF and eBPF instructions correspond to 8 bytes, and not all bytes can take arbitrary values, because some of them have restrictions due to verification or translation. Only `imm` and `k` can be used in an *unconstrained* manner, and therefore we will only use these fields for EPF purposes—this corresponds to every other 4 bytes in the {c, e}BPF instruction array. Other fields can be *partially-controlled* or are only controllable iff the BPF program is constructed in a specific way. Hence, more fine-grain control is also possible (see Figure 1), but this is something that we do not explore for mounting an EPF attack.

BPF-code Spraying Although an attacker can control parts of a BPF program's content (both in the case of translated eBPF code and the original cBPF code), they still need to create *reliable references* to such BPF instructions. This can be achieved by *spraying*: i.e., saturating kernel space with BPF programs, and, as a result, a randomly-chosen location will likely contain (malicious) BPF code [67]. Both cBPF and eBPF code are page-aligned, and hence no additional care is needed to locate memory offsets within a page. In the Linux kernel, all allocated objects can be found in the `physmap` region, which maps the whole RAM.

Figure 3: A snippet of the same eBPF code interpreted from different offsets (above vs. below).

During our experimental evaluation (§6.2), we discovered that `setsockopt` is the most effective spraying apparatus: it can be used to saturate ≈80% of all RAM, if the attack can utilize both the translated eBPF code and the original cBPF code, or ≈40% when only one of the two can be used.

BPF Interpreter The interpreter function receives two arguments: a *context* pointer and an (eBPF) instruction pointer. Context is the input to the BPF program—for example, in packet filtering, context points to the respective network packet. The value of the context pointer is loaded onto eBPF register `R1`. (The interpreter will not use this value unless it is used by the BPF program.) The second argument points to the eBPF code that is assumed to be verified. Hence, the interpreter does not validate any (eBPF) operation during execution. Because of its expressiveness, and relatively few side-effects, the interpreter is useful, as a code-reuse target as we will demonstrate in our EPF v1 (BPF-Reuse) attack.

4.2 EPF v1 (BPF-Reuse)

As we explained earlier (§4.1), an attacker can reliably control a large portion of kernel space, by (ab)using the BPF infrastructure, but they can only inject arbitrary values on every *other* 4-byte word—and thus cannot encode traditional code-reuse (e.g., ROP) payloads, which would require controlling 8 consecutive bytes in 64-bit platforms, like x86-64. However, the Linux kernel contains a powerful subsystem that can be leveraged to perform malicious computations with relatively sparse memory control: i.e., the *BPF interpreter*.

Figure 2 depicts the memory layout during an EPF v1 (BPF-Reuse) attack. First, the attacker *encodes* their payload in *valid* BPF programs, and sprays them in kernel space. Then, using a memory corruption vulnerability, they overwrite a code pointer and redirect the control flow to the BPF interpreter. By carefully choosing the respective code-pointer value, the attacker can control the *context* in which the (overwritten) code pointer will be invoked, thereby allowing them to specify an *arbitrary eBPF instruction* to start the interpretation from (i.e., by selecting/controlling the second argument of the invoked function), when the control flow is redirected to the BPF interpreter. Finally, since the attacker has sprayed BPF programs in kernel memory, they can easily find their payload in the direct mapping (i.e., `physmap`) region, with high probability (see Section 6.2).

Eventually, the problem becomes: *Can the attacker embed malicious eBPF code inside a benign BPF program, which escalates their privilege upon execution?*

```
R0 = R0 ^ R0    /* R0 =     {0} */
R1 = R1 / R1    /* R1 =     {1} */
R0 += R1        /* R0 =     {1} */
R0 += R0        /* R0 =    {10} */
R0 += R0        /* R0 =   {100} */
R0 += R1        /* R0 =   {101} */
R0 += R0        /* R0 = {1010} */
R0 += R1        /* R0 = {1011} */
```

Listing 1: Register-only BPF instructions that load the value 11 (`1011` in binary) onto register `R0`. (Values in comments are shown in binary.)

The answer is *positive*. The attacker can offset the eBPF instruction pointer by, say, 4 bytes during the attack, so that the `imm` fields will be in the original locations of `code`, `dst`, `src`, and `off`. A snippet of our eBPF payload is shown in Figure 3. When executed normally, it is an array of instructions that load immediates. However, when executed from a +4b offset, the semantics of that instruction stream are completely different. In what follows, we explain how one can perform different computations under the aforementioned model.

Arithmetics and Register Manipulation All instructions that do not use immediates, such as those moving values, or doing arithmetic, between registers, can be embedded into the `imm` field, as unused immediates are ignored by the interpreter. All *unaligned* instructions shown in Figure 3 do not use any immediates. If the attacker wishes to use constant values, they first need to load the constant into a register, and then perform the respective operation(s) with registers. Loading *arbitrary constants* into a register, without using the `imm` field, can be done solely with arithmetic operations.

Despite not being able to encode arbitrary numbers into `imm`, the attacker can load a non-zero value into a register. Then, say, `div` a register by itself, which leaves the value `0x1` in the register, or `xor` a register with itself, resulting in a `0x0` value. Similar to Listing 1, starting from `R0 = 0x0` and `R1 = 0x1`, any constant can be obtained by repeatedly adding `R0` to itself, and (optionally) adding `R1` to `R0`.

Control Flow Register-based comparisons and jumps can be encoded as usual because they do not use immediates—so conditionals can be done easily. *Looping* is also possible in malicious eBPF code. `off` field is a *signed* 16-bit value: it is signed because the `ld` and `st` instructions can use negative offsets for memory accesses. The eBPF verifier statically checks for non-negative offsets in jumps, but at runtime the interpreter does not check for this property (§2.2).

Memory Access `ld` and `st` instructions can directly access the whole kernel memory. Normally this does not create a security issue because of the (register) range analysis performed by the static checker, which ensures that no such instruction will be performed on a register that (potentially) points outside of the desired bounds. However, similar to branching offsets, this property not enforced at runtime by the interpreter.

By combining the three aforementioned techniques, the attacker can embed malicious eBPF code inside a benign BPF program. With a piece of unverified eBPF code operating in kernel memory space, there are different ways to escalate privilege. In our exploits, we chose to locate `init_task`, a global symbol that is placed in the same linked list with all `task_structs`. Then, we iterate over all processes until we find the attacking process. Lastly, we overwrite the credentials of the attacking process with those of `init_task`, giving the attacking process the highest privilege.

Combining eBPF and cBPF When spraying BPF programs, the RAM can be filled close to full, but not with only the translated eBPF code. The original cBPF code, the user process, the sockets created to attach the BPF programs to, and the JITed eBPF (if enabled), also reside in physical memory and compete for space. It is possible to encode payloads in both eBPF code and in the original cBPF code, and greatly increase the effectiveness of spraying, which translates to a higher probability of successful exploitation.

The problem is that although the allocations are always page-aligned, eBPF code and cBPF code do not start from the same offset *within* their respective pages. For example, in Linux v5.10, the eBPF code starts at a `0x38`b offset, within its page, while cBPF code starts at the beginning of the page. This can be mitigated by using a technique similar to a `NOP`-sled [30]: the attacker encodes malicious eBPF instructions inside benign instructions by offsetting the starting point by 4 bytes. Hence, in the example above, the attacker can start the malicious eBPF code with 7 (`=0x38/8`) instructions that are `xor R9, R9`; these instructions have no effect on the malicious functionality. However, if the attacker now aims at byte `0x3c` within a random page, they will be hitting a memory location that contains either eBPF or cBPF code.

4.3 EPF v2 (BPF-ROP)

Although the memory layout of eBPF code forbids the attacker from embedding 64-bit pointers, it is still possible to do so on 32-bit platforms. This facilitates ROP [100] (or, in general, code-reuse [35, 40, 57]) attacks. We will demonstrate this on x86 by introducing EPF v2 (BPF-ROP). To initiate, say, a ROP attack, the attacker usually needs to overwrite a code pointer with a stack-pivoting gadget [93], moving the stack pointer to the payload, which is an array of code pointers pointing to other gadgets. An immediate strategy would be to use the eBPF code as the payload. Doing so would encounter two challenges: (a) not every 4 bytes can encode return addresses, and (b) such a stack would be read-only.

Since the attacker has control only over every 4 other bytes, some gadgets would make `%esp` point to the gaps in between. If the execution hits a `ret` instruction in this case, it will crash the kernel and terminate the attack. More specifically, if a gadget does not move the stack pointer by +4X bytes, where X is an *odd* number, then the attack will fail.

Access Type	Regular Data	BPF Programs	
		Aligned	Unaligned
Normal Access	Allowed	BPF-ISR	BPF-ISR
BPF Execution	BPF-NX	Allowed	BPF-CFI

Table 1: How regular memory accesses and BPF-code fetches are hardened by our defenses.

To overcome this, EPF v2 filters gadgets based on how they move the stack pointer. Namely, the attacker can use gadgets that have the form: `...; pop %reg; ret`, where the `pop` instruction offsets the return address to a correct location.

EPF v2 divides a ROP (or code-reuse) attack into two stages. The first stage uses stack lifting gadgets only, and bootstraps a new ROP payload for the second stage, at a writable memory area (preferably in the original stack). This strategy requires simpler semantics for the more restricted first stage ROP, which makes gadget-finding much easier. For example, the following is a set of gadgets that can be used to bootstrap a new ROP payload: (1) `pop %edx; pop %ecx; pop %ebx; ret`, (2) `dec %eax; pop %ebp; ret`, and (3) `mov %ecx, (%eax); pop %ebx; ret`. Assuming `%eax` points to a writable region, gadget (1) can load a constant in the BPF filter into register `%ecx`, and gadget (2) can move that value to the future stack; then gadget (3) moves the future stack pointer further down for the next value. In the second stage, the attacker can use arbitrary gadgets. There are a lot of options once the attacker reaches this point. A common practice is to invoke `commit_creds(prepare_kernel_cred(0))` [88]. The ROP representation of the above, in x86 Linux, is just three addresses on the stack: a gadget to clear `%eax`, the address of `prepare_kernel_cred`, and the address of `commit_creds`. (This would not be able to execute during the first stage, as it would require a writable stack.)

5 Hardening BPF against EPF-style Attacks

5.1 Goals and Objectives

Our attacks (EPF v1 and v2) have demonstrated design weaknesses in the BPF infrastructure on Linux. Specifically, they reveal the weak separation between BPF programs and regular kernel memory: arbitrary kernel objects can be used in lieu of eBPF instructions in BPF-Reuse (EPF v1); and BPF code is used as native code pointers in BPF-ROP (EPF v2). More importantly, the problem is exacerbated by the weak runtime checks performed by the eBPF interpreter.

Taking inspiration from hardening native code, we propose to enforce the following properties (shown in Table 1):

① **Prevent regular kernel data from being used as eBPF instructions.** From the perspective of the BPF execution engine, this is analogous to data being non-executable [6]. Without this guarantee, we cannot realistically enforce any property on BPF programs, since they can be counterfeited using regular data. This property stops BPF-Reuse that utilizes the original cBPF code.

② **Ensure that BPF execution starts from benign addresses.** This is similar to control-flow integrity (CFI) on native code [29]. Since all BPF jumps have hard-coded offsets, no indirect branching is possible. The only way to divert the intended BPF control flow is to start from an unintended/unaligned address. This property stops BPF-Reuse that utilizes eBPF code.

③ **Prevent eBPF instructions from being used as regular (control) data.** By isolating BPF programs from regular kernel data, regardless of the amount of BPF programs created, the kernel cannot be misled to access malicious payloads that are embedded inside BPF programs. This property stops BPF-ROP attacks.

Besides the above security goals, we also want the respective code changes to incur negligible runtime overhead.

```
u64 bpf_interpreter(struct bpf_prog *prog)
{
    ...
    enter_bpf_mode();
    check_bpf_cfi(prog);
    initialize_context();
    mask = prog->mask;
    ...
    insn = prog->insns;
select_insn:
    tmp_insn = *insn;
    check_bpf_nx(insn);
    check_bpf_mode();
    tmp_insn = unmask(tmp_insn, mask);
    execute_bpf_insn(tmp_insn);
    if (finished) {
        goto done;
    }
    else {
        insn++;
        goto select_insn;
    }
done:
    leave_bpf_mode();
    return result;
}
```

Listing 2: Pseudocode of the BPF interpreter, instrumented with our defenses against EPF.

5.2 Design

Shown in Listing 2 is a pseudocode implementation of the BPF interpreter; colored lines correspond to our defenses.

BPF-NX To achieve objective ①, we reserve a region in the kernel's address space that is used *exclusively* for allocating BPF programs (not cBPF code, as it is not interpreted).

The interpreter can tell the difference between eBPF code and normal data by just checking an address range (ln. 12[1]). The check is performed for every eBPF instruction load, which is analogous to how a CPU enforces that the instructions are fetched from an executable page. Such frequent checking is required because in a code-reuse attack, the attacker might branch to the middle of the interpreter and bypass any one-time check outside of the main execution loop. In such scenarios, we still need the interpreter to reject instructions from invalid memory ranges.

BPF-CFI Objective ② is essentially defending against code-reuse attacks in BPF programs. BPF-CFI is challenging since we cannot simply check the alignment of the eBPF instructions. Although our attack uses unaligned eBPF instructions for simplicity, it can still be dangerous to even start executing aligned eBPF instructions from the middle of the eBPF code, because the static verifier's security guarantees only hold for executions starting from the beginning of the eBPF code. So the interpreter needs to make sure that the instruction array begins from the correct position in the eBPF code. We add such a check at the beginning of the interpreter (ln. 5).

However, this is not enough. The integrity of the control-flow [29] of the whole interpreter (function) is also necessary. Otherwise, the security check can just be skipped by leveraging a code-reuse attack that starts the execution from the "middle" of the interpreter. Kernel-level CFI [47, 85] incurs some non-negligible overhead, because it protects all code, and cannot be scaled down to protect selected parts of the kernel. Instead, we introduce a *sentinel* variable that is not corruptible by normal execution. The sentinel is used to ensure the control-flow integrity of the interpreter. The sentinel is set at the start (ln. 4), indicating that the interpreter is properly executed; at the end (of the interpretation) the sentinel is cleared (ln. 24). The interpreter checks the sentinel on each eBPF instruction fetch (ln. 13) to ensure control-flow integrity. If the control flow enters the interpreter without going through the correct entry point (i.e., the beginning of the function), it will be caught before any eBPF instruction is executed.

BPF-ISR To achieve objective ③, an obvious solution is to make use of the separation we have done in BPF-NX: whenever the rest of the kernel wants to access normal data, check whether the data lives in a BPF region.

However, it would be inefficient to instrument every memory access the kernel makes. Instead, we adopt the idea of ISR (Instruction Set Randomization), a defense originally designed to counter code injection [33, 65, 101]. ISR is suitable in this case as it is very easy to implement in a software-based interpreter. Every time the attacker tries to allocate a BPF program, the in-memory representation is chosen randomly from one of 2^{32} possibilities. Under normal BPF execution, the mask is extracted at the beginning (ln. 7), then used to unmask every instruction during interpretation (ln. 14).

[1] All line numbers in this section refer to Listing 2.

The attacker can no longer benefit from tricking kernel code into accessing BPF programs as normal memory because the content is randomized and unpredictable.

5.3 Implementation

We implemented our defenses in x86-64 Linux.

BPF-NX In 64-bit Linux, there exist gaps in the kernel's address space that are not used. We reserve a 512GB *unused* region exclusively for BPF programs (`struct bpf_prog`). Originally, the BPF programs are allocated using `vmalloc`, which is a wrapper function around `__vmalloc_node_range`, whose parameters indicate the target range in which the memory is allocated from. `vmalloc` uses a *fixed* range, specified by `VMALLOC_START` and `VMALLOC_END`. To allocate in our reserved eBPF region, we add our own wrapper `bpf_vmalloc` that calls `__vmalloc_node_range` with the proper range. A small change to the page fault handler is also needed, because of the lazy propagation of changes in kernel page tables. The BPF program region needs to be handled similarly to `vmalloc`, where its page table entries are populated on-fault.

BPF-CFI We implement the sentinel variable using the `AC` flag in `RFLAGS`. This flag is the switch for SMAP: turning off the flag allows the CPU in supervisor mode to access user data. We assume that the interpreter itself is benign and does not need the protection against unintended user-space memory accesses. The sentinel variable is set by "turning off" SMAP, and cleared by turning it on. This way, the sentinel variable can easily be checked by an access to a user memory page. During kernel initialization, a dedicated memory page is marked as a user-mode page, and reserved for the access check. At the beginning of the interpreter, we execute the instruction `clac` to disable SMAP. Whenever the interpreter fetches an eBPF instruction, it also reads from the user-mode page, verifying that SMAP is indeed disabled. And at the end, the interpreter executes `stac` to re-enable SMAP.

For the check re: the starting point of the eBPF instruction array, we added 8 `0xff` bytes in `struct bpf_prog` as a *magic* number, which eBPF instructions cannot forge. By checking that this exists at the correct position inside the BPF program header, we assert that it is the right starting point.

BPF-ISR To implement ISR in the eBPF interpreter, we add a new field `mask` in `struct bpf_prog` to store the mask value for each BPF program. After a BPF program is initialized, all the pages are changed to read-only. Right before the permission change, we choose a random 4-byte value as mask, and `xor` the `imm` field in every eBPF instruction with our mask. This ensures that the memory content can not be arbitrarily chosen by the attacker. Since original cBPF code is also stored inside kernel memory, naturally it needs to be masked too. We do not mask the rest of the eBPF instructions because the attacker does not have much control over them. Lastly, the interpreter unmasks each instruction in the stack during the execution of the eBPF instructions.

CVE	Vulnerability Type	Context	Method
CVE-2021-43267	Heap overflow	Process	EPF v1
CVE-2017-7308	Heap overflow	Process	EPF v1
CVE-2016-8655	Use-after-free	Interrupt	EPF v1
CVE-2017-7308	Heap overflow	Process	EPF v2
CVE-2017-6074	Use-after-free	Process	EPF v2
CVE-2016-8655	Use-after-free	Interrupt	EPF v2
CVE-2013-2094	Arbitrary write	Process	EPF v2

Table 2: List of vulnerabilities exploited with EPF.

6 Evaluation

To evaluate EPF (§4), and the set of defenses we developed against it (§5), we used a host armed with a 16-core 3.7GHz Intel Xeon W-2145 CPU and 64GB RAM, running Ubuntu 18.04 LTS (64-bit).

In our evaluation we focused on the following questions:

- **RQ1:** Are EPF attacks realistic?
- **RQ2:** How effective is EPF-based payload injection? How does it compare to other methods?
- **RQ3:** How much overhead does our set of defenses against EPF introduce?

6.1 Effectiveness of EPF (RQ1)

To demonstrate the feasibility of EPF-style attacks, we applied BPF-Reuse and BPF-ROP on existing Linux vulnerabilities with publicly-available exploits (see Table 2).

For BPF-Reuse (EPF v1) we chose 3 vulnerabilities. CVE-2021-43267 and CVE-2017-7308 are heap overflow bugs; our exploit dereferences an overwritten code-pointer in process context. CVE-2016-8655 is a use-after-free bug; a controlled function-pointer is dereferenced in interrupt context. Our payload loops through the linked list of *all* `task_structs` and changes the credentials on the one with the proper `pid`, using the interpreter (§4.2)—as a result, our strategy works in both contexts. This shows the expressiveness and versatility of EPF-based exploitation.

For BPF-ROP (EPF v2) we chose 4 vulnerabilities that also cover interrupt and process contexts with different types of vulnerabilities; 3 vulnerabilities happen in process context, and the respective payloads are setup to invoke `commit_creds(prepare_kernel_cred(0))` (§4.3), whereas for the interrupt context scenario, the ROP payload marks the memory page(s) that host the BPF program itself as executable, and transfers control to (x86) shellcode that in encoded as valid BPF instructions. With native shellcode embedding, we also employ the same strategy of looping through active processes and performing privilege escalation.

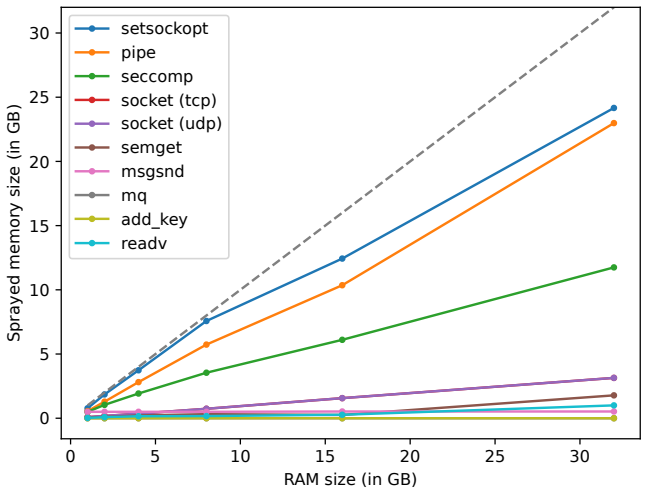


Figure 4: Effectiveness of spraying using different syscalls.

6.2 Spraying Effectiveness (RQ2)

We measured the effectiveness of EPF-based spraying and compared it to other methods. We found reasonable candidates for our comparison. Namely, we chose syscalls as spraying aids using the following three criteria:

• *Syscalls with variable-sized arguments*: In this case, we aim to find syscalls that copy variable-sized data into the kernel. This is done by filtering out syscalls that have a `const` pointer as argument, accompanied by a `size_t` or integer argument specifying the array size. This set includes `add_key` and `readv` (representing all vectorized I/O syscalls).

• *Syscalls returning writable file descriptors*: In this case, we aim to find syscalls that create file descriptors, which can later be used together with syscalls such as `write` to inject data into the kernel. As a result this set includes `pipe`, `bpf`, `socket`, and `landlock_create_ruleset`. The latter is not available in latest Debian (v11), and `bpf` is disabled for unprivileged users by default, so we exclude them.

• *Multiplexed syscalls*: In this case, we manually investigated syscalls with multiplexed arguments (e.g., `ioctl`, `fcntl`) and tried to find semantically similar syscalls for spraying. This set includes `setsockopt` and `seccomp`.

All our experiments took place on a Debian v11 (bullseye) VM, running on the benchmarking host, with its RAM size set at 1, 2, 4, 8, 16, and 32GB, respectively. Specifically, we first start a *monitor* process, which then forks a *sprayer* process and communicates with it via a set of Unix pipes. When the latter stops consuming additional memory, the monitor will spawn a new sprayer. For each method, the probability that the attacker can locate one of the sprayed objects is denoted as *success rate*. Lastly, the respective spraying method can bypass strong kernel-user isolation mechanisms (like XPFO [67]), unless we mention otherwise. As shown in Figure 4, creating cBPF (using `setsockopt` or `seccomp`) is among the most efficient methods.

setsockopt and seccomp We are using `setsockopt` with `SO_ATTACH_FILTER` to attach cBPF programs to sockets, and `SECCOMP_MODE_FILTER` to attach cBPF programs to processes. These two methods do not have quotas/limits set from the underlying kernel. In fact, if we keep allocating cBPF programs, the system will invoke the OOM-killer and try to reclaim memory at some point. If both the translated eBPF programs and the original cBPF copies are utilized (as described in Section 4.2), `setsockopt` can take up to 70% of all physical memory, while `seccomp` can take up to 34%; otherwise, the ratios reduce to 35% and 17%, respectively. In conclusion, spraying cBPF programs has a 70% or 35% success rate depending on the mode used, and the data structure allows controlling every 4 other bytes.

msgsnd The `msgget` syscall creates message queues for SysV-based IPC, and `msgsnd` adds messages to such queues. The maximum memory occupied by the messages is 500MB, regardless of the RAM size. The probability of locating sprayed content is 3%–24% (depending on the total RAM size) with continuous control over the sprayed region.

semget The `semget` syscall creates (SysV) semaphores. By generating 32K sets, each with 32K semaphores, the total limit will be reached. Each semaphore takes up 64 bytes of kernel memory, so the semaphores can consume ≈60GB of memory. But, a significant drawback of this approach is that of the 64-byte data structure only the semaphore variable can be controlled by the attacker, which is only 4 bytes. Therefore `semget` is not realistically useful.

pipe We create 20K Unix pipes for each sprayer process, and write exactly one page of content into each. This method can fill up to 60% of the total physical memory. Since the memory used to store pipes is page-aligned, this translates to 60% success rate, with complete attacker control. However, the sprayed content can be easily isolated using strong kernel-user separation mechanisms (e.g., XPFO).

mq By default, an unprivileged user can have 800KB of data stored in-kernel using POSIX message queues. The attacker will have less than 1% chance of locating the sprayed objects. Additionally, the sprayed content can also be isolated by strong kernel-user separation mechanisms, like XPFO.

socket We use the following strategy to spray with TCP sockets: for each process, create one TCP socket, send messages until it blocks, and then spawn as many processes as possible until socket creation fails. UDP sockets are different; they do not block with failed sending. Hence, we send until the occupied memory in `/proc/net/sockstat` does not change. In both cases, we are able to spray about 10% of the physical memory. This method has 10% success rate and allows for complete control over the sprayed region.

readv and add_key `readv` gives the attacker less than 2% success rate, controlling 6 out of every 8 bytes, while the total amount of memory that can be allocated by `add_key` is 20K bytes. So the success rate of the latter is less than 1%.

6.3 Hardening Overhead (RQ3)

We mainly evaluate our defenses on syscall filtering [80], socket filtering, and XDP [78] `skb` mode.

Syscall Filtering `sysfilter` [52] is an automated syscall filtering tool. It analyzes an application, creates the set of syscalls that the application needs, and then enforces it using `seccomp-bpf`. We evaluate the overhead our defenses incur on Nginx and Redis, when hardened by `sysfilter`, with no BPF-JIT, and the SSB mitigation disabled (by setting `SECCOMP_FILTER_FLAG_SPEC_ALLOW` [80]). The network requests in each test are sent over the loopback (`lo`) device. Both packages are installed from Ubuntu's repository.

To benchmark Nginx, we used `wrk` [56] with 2 running threads, each performing 128 connections, for 1 minute continuously. Nginx is configured to have 2 working processes. To maximize the time spent on BPF execution, we picked the smallest size of requested file from `sysfilter`, which is 1KB. Additionally, we manually inspected the CPU utilization, ensuring it was close to 100%. To benchmark Redis, we used `memtier` [97], spawning 2 worker threads, each with 128 clients, running for 1 minute continously. The ratio between `GET` and `SET` operations was set to 10:1, and each data object was 32 bytes. Again, we also made sure that the CPU utilization was close to 100%. Our results are shown in Figure 5. Our defenses introduce an additional 1.8% throughput decrease on Nginx, and 1.5% on Redis.

Socket Filtering We focus on the usage of socket filtering, similar to a traffic monitoring scenario. A simple traffic generator will send UDP packets in a tight loop, with a body size of 64 bytes to the DUT (device under test); and on the DUT there is a server that receives them. Both machines have 1GbE NICs, and the sending speed is tuned to be slightly higher than the processing capability of the DUT. We use small packets because large packet sizes will mask the BPF processing time.

To simulate a traffic monitor, we attach a raw socket to the ethernet interface and call `setsockopt` to attach a cBPF filter on the raw socket. We use a set of 6 different filter rules, similar to previous network monitoring studies [111], with some changes: (1) the rules are slightly modified to make sure our packets go through the maximum possible execution paths; and (2) we also modify the return instruction to always reject packets and not spend any time reading them, resulting in all overhead showing up on the receiving end.

The cBPF filters are described in the following (`PCAP`):

1. `""` (empty expression that allows everything)
2. `"ip"`
3. `"ip src net 10.116.70.0/24 and dst net 10.0.0.0/8"`
4. `"ip src or dst net 192.168.2.0/24"`
5. `"ip and udp port (10 or 11 or 12 or 13 or 14)"`
6. `"ip and (not udp port (80 or 25 or 143)) and not ip host ..."` (32 IPv4 addresses)

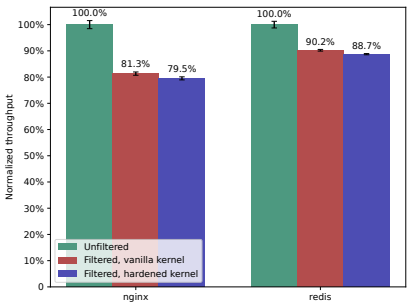


Figure 5: Normalized throughput between vanilla Linux and EPF-hardened Linux when running `seccomp`-protected Nginx and Redis.

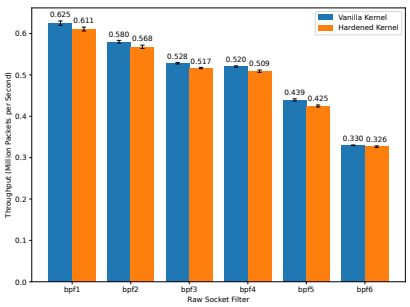


Figure 6: Traffic monitoring performance using different socket filters.

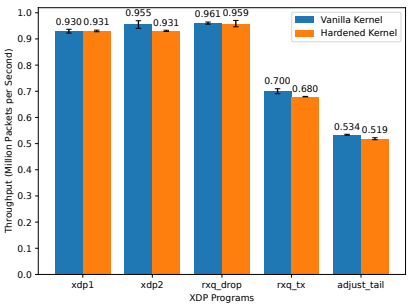


Figure 7: Throughput of XDP programs when run on vanilla kernel vs. an EPF-hardened kernel.

The results are shown in Figure 6; there is a 0.5%–3% reduction in the respective throughput.

XDP Since all cBPF programs are executed as eBPF programs, our defense affects interpreted eBPF programs too. XDP uses eBPF programs to passthrough, drop, re-transmit, or re-route incoming packets. In our experiment, we run XDP in `skb` mode, which executes eBPF programs when packet handling enters the device-agnostic part of the kernel, as the other modes require specific hardware. The experiment setup is the same as for socket filtering. To demonstrate the performance impact on eBPF programs, we run XDP programs (shown below) from the Linux source tree, similar to previous works [39, 59]. The XDP programs will intercept and run on every incoming packet generated by the UDP client.

- `xdp1`: Parse the IP header, count incoming packets and update a counter in a BPF map, then drop the packets.
- `xdp2`: Same as `xdp1`, but re-transmit the packets.
- `xdp_adjust_tail`: Change incoming packets into ICMP packets and send them back, keeping a total count in a BPF map.
- `rxq_info(drop)`: Count incoming packets for each receive queue and drop them.
- `rxq_info(tx)`: Count incoming packets for each receive queue and re-transmit them.

We tuned the traffic generator to send at a higher rate than the maximum throughput. Additionally, we pinned the NIC interrupts to one CPU core and manually verified that CPU utilization was close to 100%. The results are shown in Figure 7: we observed 0%–3% throughput degradation.

7 Discussion

BPF-CFI Considerations To implement BPF-CFI we utilize the `AC` flag, and, as we mentioned in Section 5.3, this flag also controls SMAP. We chose this approach because SMAP was designed to accommodate low-overhead switching.

However, this also means that during the execution of the BPF interpreter SMAP cannot prevent the interpreter from incorrectly accessing userspace data. At first glance this might be a security loss, but in fact the impact is very minimal. Firstly, this is a confined attack surface that is relatively easy to maintain, instead of a complicated invariance to be respected across the whole kernel codebase. Secondly, SMAP is designed to stop the kernel from incorrectly accessing user-controlled data during an attack, but the semantics of BPF programs already grant the interpreter access to user-controlled content such as BPF maps [5], context metadata, and more. In conclusion, SMAP does not provide notable security gains in the BPF interpreter context, while our defense provides better overall security by re-purposing this extension.

KASLR Bypass Although KASLR [54] is sometimes bypassed by arbitrary memory disclosure vulnerabilities, much weaker primitives exist that circumvent KASLR, including bounded memory disclosure vulnerabilities [18, 20, 44] and side-channel attacks [41, 58, 62, 76]. By spraying, our attacks can locate attacker-controlled objects *within the heap*. If the attacker deploys one of the methods to de-randomize KASLR, then they can find *where the heap is located at*. Combining these two capabilities, our attacks can work exactly the same as they would without KASLR.

8 Related Work

BPF JIT Spraying BPF JIT spraying [82, 98] is an exploitation technique that takes advantage of the BPF's JIT engine generating predictable code. By carefully crafting and spraying the JITed code, the attacker can control a piece of code in kernel context, which effectively nullifies defenses against ret2usr such as SMEP and PXN, and re-enables attacks that redirect control flow to user-controlled code—JITed BPF code. BPF-Reuse and BPF-ROP utilize the BPF program data structure itself and aim to control memory content.

Thereby turning BPF programs into a mechanism for injecting payloads for code-reuse attacks. Importantly, the targeted features and the goals of BPF JIT- vs. EPF-based spraying are completely different. Moreover, BPF JIT spraying is already defended against in recent Linux kernel [79], whereas BPF-Reuse and BPF-ROP bypass all existing isolation mechanisms, including XPFO [67], which is not yet deployed in mainline Linux.

eBPF-based Speculative Type Confusion Kirzner et al. [71] pointed out that the eBPF verifier performs extensive analyses, and safety checks, to ensure the execution of eBPF programs is sandboxed, but they did not take into account speculative execution paths. As a result, some eBPF programs deemed safe by the verifier can be vulnerable to transient execution attacks and leak confidential kernel data. The root cause is addressed by adding analyses that account for possible speculation [37]. Our attacks are possible due to a different underlying reason: BPF programs are a type of user-controlled memory object that cannot be easily isolated from the rest of kernel data, and they can be created in bulk.

Other Popular Kernel Exploitation Techniques In the post-ret2usr era, where defenses such as SMEP, SMAP, PXN, and PAN are present, kernel exploitation has evolved to allow for creating addressable payloads at exploitation time. Apart from ret2dir [67], which is discussed in Section 2.1, there are also two other popular strategies to bypass ret2usr defenses. The first strategy is careful heap manipulation that combines heap fengshui [102, 108] with elastic objects (systemized and termed by Chen et al. [43]), which results in disclosing the address of the user-controlled memory object that will become the attack payload. The strategy is used by some real-world exploits [87, 92], and takes advantage of the predictability of the heap layout, whereas EPF abuses the design of BPF functionality. The second strategy is calling functions inside the kernel, which can disable protection mechanisms. It is used by real-world exploits [75], as well as automated exploit generation frameworks [110]. The drawback is that it relies on how defense features are implemented.

After the payload is placed in kernel space, there are several ways to actually realize privilege escalation. One popular method is to overwrite the `modprobe_path` global variable [69], substituting an attacker-controlled binary to be executed with `root` privilege (instead of `/sbin/modprobe`). This is used by exploit authors and the CTF community [109, 114]. Another method is `ret2bpf` (termed by Jin et al. [63]), which is popular in ARM kernel exploits [31] because it does code-reuse using the BPF interpreter, significantly simplifying the search for code gadgets. It tricks the kernel to use attacker-controlled memory as BPF instructions, essentially doing "BPF-code injection", which notably can be defended against by BPF-NX. `ret2bpf` has similarities to BPF-Reuse, but, most importantly, it differs in the method of supplying the attack payload: `ret2bpf` requires the attacker to have the ability to create a payload in kernel space, whereas BPF-Reuse solves exactly this problem (i.e., payload injection).

9 Conclusion

In this paper, we have shown that BPF, a kernel subsystem that allows unprivileged users to push data structures, freely, onto the kernel address space, is inherently susceptible to attack-payload injection. We developed two attacks, BPF-Reuse (EPF v1) and BPF-ROP (EPF v2), and demonstrated how to inject certain payloads. Further, we showed the practicality, and effectiveness, of our attacks by combining them with real-world vulnerabilities to exploit the Linux kernel. We also developed comprehensive defenses that enforce the stronger isolation between BPF code and normal kernel data, and the integrity of BPF program execution, thwarting the abuse of the BPF infrastructure. Our defenses were evaluated on tasks that result in heavy BPF usage, and were shown to have negligible overhead.

Availability

Our prototype implementation of BPF-{NX, CFI, ISR} and the exploits we ported to EPF are available at:
https://gitlab.com/brown-ssl/epf

Acknowledgments

We thank our shepherd, Michael Le, and the anonymous reviewers for their valuable feedback. We also thank Alexander Gaidis for providing comments on earlier drafts of our paper. This work was supported in part by the CIFellows 2020 program, through award CIF2020-BU-04, and the National Science Foundation (NSF), through award CNS-2238467. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, NSF, or CRA.

References

[1] A safer playground for your Linux and Chrome OS renderers. https://blog.chromium.org/2012/11/a-safer-playground-for-your-linux-and.html.

[2] ARM Cortex-A Series Programmer's Guide for ARMv8-A. https://developer.arm.com/documentation/den0024/a/BABCEADG.

[3] bpf-helpers(7) – Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[4] bpf(2) – Linux manual page. https://man7.org/linux/man-pages/man2/bpf.2.html.

[5] eBPF maps. https://www.kernel.org/doc/html/latest/bpf/maps.html.

[6] Kernel Self-Protection. https://www.kernel.org/doc/html/latest/security/self-protection.html#executable-code-and-read-only-data-must-not-be-writable.

[7] Learn the architecture – AArch64 memory model. https://developer.arm.com/documentation/102376/0100/Permissions-attributes.

[8] libbpf. https://github.com/libbpf/libbpf.

[9] Mozilla wiki – Security/Sandbox/Seccomp. https://wiki.mozilla.org/Security/Sandbox/Seccomp#Use_in_Gecko.

[10] Seccomp security profiles for Docker. https://docs.docker.com/engine/security/seccomp/.

[11] sysfs – The filesystem for exporting kernel objects. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html.

[12] syzbot. https://syzkaller.appspot.com/openbsd.

[13] syzkaller – kernel fuzzer. https://github.com/google/syzkaller.

[14] The /proc Filesystem. https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[15] XDP – eXpress Data Path. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/index.html.

[16] CVE-2017-16996. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16996, November 2017.

[17] CVE-2017-17853. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17853, December 2017.

[18] CVE-2017-17864. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17864, December 2017.

[19] CVE-2019-7308. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7308, February 2019.

[20] CVE-2020-25662. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-25662, September 2020.

[21] CVE-2021-45402. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45402, December 2020.

[22] CVE-2021-31829. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31829, April 2021.

[23] CVE-2021-32606. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-32606, May 2021.

[24] CVE-2021-33034. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33034, May 2021.

[25] CVE-2021-3444. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3444, March 2021.

[26] CVE-2021-3490. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490, April 2021.

[27] CVE-2021-3612. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3612, June 2021.

[28] CVE-2021-42008. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42008, October 2021.

[29] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

[30] Periklis Akritidis, Evangelos P Markatos, Michalis Polychronakis, and Kostas Anagnostakis. Stride: Polymorphic Sled Detection through Instruction Sequence Analysis. In *IFIP International Information Security Conference (SEC)*, pages 375–391, 2005.

[31] Brandon Azad. An iOS hacker tries Android. https://googleprojectzero.blogspot.com/2020/12/an-ios-hacker-tries-android.html.

[32] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security Symposium (SEC)*, pages 971–988, 2022.

[33] Elena Gabriela Barrantes, David H Ackley, Stephanie Forrest, Trek S Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, 2003.

[34] Ashish Bijlani and Umakishore Ramachandran. Extension Framework for File Systems in User Space. In *USENIX Annual Technical Conference (ATC)*, pages 121–134, 2019.

[35] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Asia Symposium on Information, Computer and Communications Security (ASIA CCS)*, pages 30–40, 2011.

[36] Daniel Borkmann. bpf: add generic constant blinding for use in jits. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4f3446b.

[37] Daniel Borkmann. BPF and Spectre: Mitigating transient execution attacks. https://github.com/gojue/ebpf-slide/blob/master/eBPF_advanced/eBPF-Summit-2021-BPF-and-Spectre-Daniel-Borkmann-Final.pdf.

[38] Brendan Gregg. Linux Extended BPF (eBPF) Tracing Tools. https://www.brendangregg.com/ebpf.html.

[39] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 973–990, 2020.

[40] Bugtraq. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63.

[41] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break it, Fix it, Repeat. In *ACM Asia Symposium on Information, Computer and Communications Security (ASIA CCS)*, pages 481–493, 2020.

[42] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating Seccomp Filter Generation for Linux Applications. In *ACM Cloud Computing Security Workshop (CCSW)*, pages 139–151, 2021.

[43] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A Systematic Study of Elastic Objects in Kernel Exploitation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1165–1184, 2020.

[44] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[45] Jonathan Corbet. A JIT for packet filters. https://lwn.net/Articles/437981/.

[46] Jonathan Corbet. BPF: the universal in-kernel virtual machine. https://lwn.net/Articles/599755/.

[47] Jonathan Corbet. Control-flow integrity in 5.13. https://lwn.net/Articles/856514/.

[48] Jonathan Corbet. Reconsidering unprivileged BPF. https://lwn.net/Articles/796328/.

[49] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475/.

[50] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[51] Debian Documentation Project. Linux disables unprivileged calls to bpf() by default. https://www.debian.org/releases/stable/amd64/release-notes/ch-information.en.html#linux-unprivileged-bpf.

[52] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 459–474, 2020.

[53] Eric Dumazet. x86: bpf_jit_comp: secure bpf jit against spraying attacks. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=314beb9.

[54] Jake Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[55] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 443–458, 2020.

[56] Will Glozer. wrk – a HTTP benchmarking tool. https://github.com/wg/wrk.

[57] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*, pages 575–589, 2014.

[58] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM Conference on Computer and Communications Security (CCS)*, pages 368–379, 2016.

[59] 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 *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 54–66, 2018.

[60] Gianluca Insolvibile. The Linux Socket Filter: Sniffing Bytes over the Network. *Linux Journal*, 86:53, 2001.

[61] IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[62] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM Conference on Computer and Communications Security (CCS)*, pages 380–392, 2016.

[63] Xingyu Jin and Richard Neal. The Art of Exploiting UAF by Ret2bpf in Android Kernel. https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf.

[64] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, 2003.

[65] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, 2003.

[66] Vasileios P Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 2015.

[67] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium (SEC)*, pages 957–972, 2014.

[68] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *USENIX Security Symposium (SEC)*, pages 459–474, 2012.

[69] Linux Kernel. Documentation for /proc/sys/kernel/. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#modprobe.

[70] Linux Kernel. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[71] Ofek Kirzner and Adam Morrison. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *USENIX Security Symposium (SEC)*, pages 2399–2416, 2021.

[72] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[73] Karsten König. Exploit for CVE-2019-5596. https://www.exploit-db.com/exploits/47829.

[74] Andrey Konovalov. Exploit for CVE-2017-1000112. https://www.exploit-db.com/exploits/47169.

[75] Andrey Konovalov. Exploit for CVE-2017-7308. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[76] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 309–321, 2020.

[77] LEXFO. Exploit for CVE-2017-11176. https://www.exploit-db.com/exploits/45553.

[78] Linux Kernel. AF_XDP. https://www.kernel.org/doc/html/latest/networking/af_xdp.html.

[79] Linux Kernel. Documentation for /proc/sys/net/. https://www.kernel.org/doc/html/latest/admin-guide/sysctl/net.html#bpf-jit-harden.

[80] Linux Kernel. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[81] LLVM Project. LLVM 3.7 Release Notes. https://releases.llvm.org/3.7.0/docs/ReleaseNotes.html#non-comprehensive-list-of-changes-in-this-release.

[82] Keegan McAllister. Attacking hardened Linux systems with kernel JIT spraying. https://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html.

[83] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter Conference*, 1993.

[84] David Miller. BPF Verifier Overview. https://www.spinics.net/lists/xdp-newbies/msg00185.html.

[85] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. DROP THE ROP: Fine-grained Control-flow Integrity for the Linux Kernel. *Black Hat Asia*, 2017.

[86] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and Verification in the Field: Applying Formal Methods to BPF Just-In-Time Compilers in the Linux Kernel. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 41–61, 2020.

[87] Andy Nguyen. CVE-2021-22555: Turning \x00\x00 into 10000$. https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html.

[88] Andy Nguyen. Exploit for CVE-2021-22555. https://www.exploit-db.com/exploits/50135.

[89] PaX Team. Better kernels with GCC plugins. https://lwn.net/Articles/461811/.

[90] PaX Team. UDEREF/amd64. http://grsecurity.net/pipermail/grsecurity/2010-April/001024.html.

[91] PaX Team. UDEREF/i386. http://grsecurity.net/~spender/uderef.txt.

[92] peter@haxx.in. Exploit for CVE-2021-43267. https://haxx.in/posts/pwning-tipc/.

[93] Aravind Prakash and Heng Yin. Defeating ROP through Denial of Stack Pivot. In *Annual Computer Security Applications Conference (ACSAC)*, pages 111–120, 2015.

[94] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, pages 563–577, 2020.

[95] Tim Rains, Matt Miller, and David Weston. Exploitation Trends: From Potential Risk to Actual Risk. In *RSA Conference*, 2015.

[96] rebel. Exploit for CVE-2016-8655. https://www.exploit-db.com/exploits/47170.

[97] Redis Labs. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.

[98] Elena Reshetova, Filippo Bonazzi, and N Asokan. Randomization Can't Stop BPF JIT Spray. In *International Conference on Network and System Security (NSS)*, pages 233–247, 2017.

[99] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance Implications of Packet Filtering with Linux eBPF. In *International Teletraffic Congress (ITC)*, pages 209–217, 2018.

[100] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[101] Kanad Sinha, Vasileios P Kemerlis, and Simha Sethumadhavan. Reviving Instruction Set Randomization. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 21–28. IEEE, 2017.

[102] Alexander Sotirov. Heap Feng Shui in JavaScript. *Black Hat Europe*, 2007.

[103] Alexei Starovoitov. bpf: introduce BPF_JIT_ALWAYS_ON config. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=290af86629.

[104] Alexei Starovoitov. bpf: split eBPF out of NET. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f89b7755f517cdbb755d7543eef986ee9d54e654.

[105] Alexei Starovoitov. tracing, perf: Implement BPF programs attached to kprobes. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2541517c32be2531e0da59dfd7efc1ce844644f5.

[106] SUSE Support. Security Hardening: Use of eBPF by unprivileged users has been disabled by default. https://www.suse.com/support/kb/doc/?id=000020545.

[107] Qualys Research Team. Sequoia: A deep root in Linux's filesystem layer (CVE-2021-33909). https://www.qualys.com/2021/07/20/cve-2021-33909/sequoia-local-privilege-escalation-linux.txt.

[108] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards Automated Heap Feng Shui. In *USENIX Security Symposium (SEC)*, pages 1647–1664, 2021.

[109] willsroot. CVE-2022-0185 - Winning a $31337 Bounty after Pwning Ubuntu and Escaping Google's KCTF Containers. https://www.willsroot.io/2022/01/cve-2022-0185.html.

[110] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards Facilitating Exploit Generation for Kernel Use-after-free Vulnerabilities. In *USENIX Security Symposium (SEC)*, pages 781–797, 2018.

[111] Zhenyu Wu, Mengjun Xie, and Haining Wang. Swift: A Fast Dynamic Packet Filter. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 279–292, 2008.

[112] Fenghua Yu. Enable SMEP CPU Feature. https://lore.kernel.org/lkml/1305581685-5144-1-git-send-email-fenghua.yu@intel.com/.

[113] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux Kernel. In *USENIX Security Symposium (SEC)*, pages 3201–3217, 2022.

[114] Xiaochen Zou and Zhiyun Qian. CVE-2022-27666: Exploit esp6 modules in linux kernel.

Translation Pass-Through for Near-Native Paging Performance in VMs

Shai Bergman
Technion

Mark Silberstein
Technion

Takahiro Shinagawa
University of Tokyo

Peter Pietzuch
Imperial College London

Lluís Vilanova
Imperial College London

Abstract

Virtual machines (VMs) are used for consolidation, isolation, and provisioning in the cloud, but applications with large working sets are impacted by the overheads of memory address translation in VMs. Existing translation approaches incur non-trivial overheads: (i) nested paging has a worst-case latency that increases with page table depth; and (ii) paravirtualized and shadow paging suffer from high hypervisor intervention costs when updating guest page tables.

We describe *translation pass-through* (TPT), a new memory virtualization mechanism that achieves near-native performance. TPT enables VMs to control virtual memory translation from guest-virtual to host-physical addresses using one-dimensional page tables. At the same time, inter-VM isolation is enforced by the host by exploiting new hardware support for physical memory tagging in commodity CPUs.

We prototype TPT by modifying the KVM/QEMU hypervisor and enlightening the Linux guest. We evaluate it by emulating the memory tagging mechanism of AMD CPUs. Our conservative performance estimates show that TPT achieves native performance for real-world data center applications, with speedups of up to 2.4× and 1.4× over nested and shadow paging, respectively.

1 Introduction

Virtualization plays a central role in cloud stacks. Many academic and industry efforts strive to bring its performance closer to that of native (bare-metal) execution [19, 23, 27, 29, 37, 51, 55, 68]. Nevertheless, memory address translation in virtual machines (VMs) introduces non-trivial performance overheads. Worse, these overheads are expected to grow as applications move to larger working set sizes [26, 45], and architectures evolve to use deeper page tables to support more physical memory [1].

Memory translation in VMs (also known as guests) is performed using one of two approaches, each with its own benefits and drawbacks. In *nested paging* (see Fig. 1a), as supported by Intel EPT [51] and AMD nPT [19], VMs self-manage page tables without involving the hypervisor (also known as the host). Nested paging, however, introduces overheads during address translation: it virtualizes guest physical addresses by combining guest page tables with an additional nested page table controlled by the hypervisor. This results in up to 6× more page table entry references than a native system [19] – the MMU must issue up to 24 memory accesses to the page tables, as opposed to 4 in a native system.

In contrast, *shadow paging* (see Fig. 1b) achieves near-native translation performance. However, the guest page table management becomes costly: the hypervisor synchronizes each guest page table with a host (or shadow) page table, which directly translates guest virtual addresses (GVAs) to host physical addresses (HPAs). This avoids the translation overheads of nested paging, but introduces expensive VM exceptions to keep the page tables synchronized — often in an application's critical path.

Despite sophisticated optimizations in today's systems, such as lazy page table shadowing [61] and partial walk caches [32], we observe that workloads see up to 2.4× and 1.4× slowdowns due to nested and shadow paging, respectively (see §6). These overheads are expected to grow in future systems: applications with larger working set sizes [26, 45] will have higher TLB miss rates; emerging workloads such as function-as-a-service (FaaS) and Kata containers [34] rely heavily on process creation inside VMs, adding to page table management overheads; and upcoming CPUs will feature deeper page table hierarchies, resulting in quadratic increases in nested page table traversal overheads [1].

We explore a new approach to memory address translation, *translation pass-through (TPT)*, which enables near-native virtual memory performance in VMs. With TPT, VMs directly control translations to their assigned physical memory, without the extra level of indirection of nested paging, and without the hypervisor interventions of shadow paging during guest page table modifications. TPT is enabled by new functionality in commodity CPUs for physical memory protection using *memory tags*, e.g., in AMD SEV-SNP [60] to support confidential computing features [22]. Our key observation is that this new type of physical memory protection can be leveraged

by hypervisors to efficiently enforce memory isolation between VMs, while allowing the VMs to manage direct guest-virtual to host-physical address translation (see Fig. 1c). Thus, TPT offers a new, more efficient point in the design space across hardware-virtualized, paravirtualized, and shadow virtual memory management.

TPT's gains in memory translation performance come from the fact that one-dimensional page walks in guest VMs, combined with hardware memory protection checks, are faster than the two-dimensional page walks using nested paging. Prior work [6, 7, 16] has shown that the overheads of tag checks can be hidden by performing them in parallel with memory accesses and translation. Recent performance results on AMD SEV-SNP CPUs with physical memory tags [60] corroborate the low-performance overhead for real-world workloads. In contrast, a nested page table walk requires extra steps that are inherently sequential, making it harder to optimize.

To realize TPT, we make the following contributions:

(1) VM isolation with hardware memory protection. TPT leverages MMU support to maintain the host's physical memory frame permissions using tags. By setting per-VM frame tags, we can safely allow VMs to manage direct guest-physical-to-host-physical page tables: the hypervisor ensures that a VM can only access host frames assigned to it, regardless of the host physical addresses in the VM's page tables ("Hypervisor" and "HW" layers in Fig. 1c). Existing AMD CPUs with SEV-SNP already support the host frame permissions we need for TPT; we cannot use SEV-SNP as-is because frame tags are coupled with nested paging and expensive memory encryption, but we would require only simple hardware changes: adding two registers to configure TPT, and enabling the frame tag functionality separately from the rest of SEV-SNP.

(2) Selective user-space translation. Enabling TPT for an entire VM would require a fundamental redesign of the boot process, memory management, and I/O in the guest OS. Fortunately, TPT's performance benefits are largest for user-space applications with large working sets, but are less so for small working-set applications or kernel-space (see §3). The guest OS thus enables TPT only in user-space execution of some processes, which are dynamically identified to take advantage of it. We achieve this by introducing a new type of *TPT page table* with GVA-to-HPA translations that are checked against the VM's host frame permissions, whereas guest kernel code and other non-TPT-enabled processes use the traditional *non-TPT page table* (similar to how PTI works [58]). This supports incremental deployments in which TPT and non-TPT processes and VMs co-exist on a host and with minimal guest OS changes, which simplifies the deployment of TPT.

(3) Hypervisor-compatible extensions. We describe a design for TPT that is compatible with existing hypervisors. TPT only requires modest changes to the KVM interface: it exposes the physical memory map to enlightened guests, and

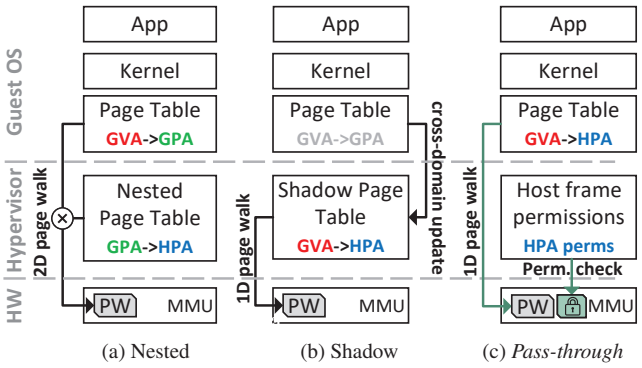


Figure 1: Existing (nested, shadow) and proposed pass-through paging approaches (GVA/GPA mean guest virtual/physical address; HVA/HPA mean host virtual/physical address; PW means page walk.)

extends the guest pvops backend in Linux to seamlessly incorporate the extra TPT page tables. Our design permits the hypervisor to retain control over guest physical memory without introducing performance penalties: the hypervisor can use existing memory ballooning techniques, and can forcibly reclaim host frames from uncooperative VMs. To support host frame reclamation and VM migration, the guest OS keeps a pair of synchronized TPT and non-TPT page tables, which we call *dual page tables*, for each TPT-enabled process; using pvops in the guest OS keeps synchronization transparent and with low overhead. Since a dual page table is always kept in sync, the hypervisor can force any guest process to utilize its non-TPT page table while a host frame reclamation or VM migration is underway.

We implement TPT using a Linux guest and KVM/QEMU hypervisor. We evaluate our TPT prototype using a commodity x86–64 CPU — which does not perform any host frame permission check, but can execute applications much larger than a traditional CPU simulator —, and assume an optimized MMU implementation that executes permission checks in parallel with page table traversal. We also model the performance of a naive MMU implementation where operations are executed in sequence by injecting additional delays in page table walks, and discuss how both approaches reasonably model the overheads that we should expect from a hardware implementation such as is contained in SEV-SNP.

Our results show that an optimized TPT implementation achieves native performance, and is 2× and 1.2× faster than nested paging and shadow paging, respectively, on a PageRank benchmark. Even with a naive MMU implementation, TPT exhibits a geometric mean slowdown of only 3% over native execution for a series of typical cloud workloads, including Memcached and kernel compile.

The TPT implementation is available as open source software at https://github.com/acsl-technion/TPT.

2 VM Address Translation

We discuss the properties of current memory virtualization approaches (§2.1) and motivate the opportunities offered by new hardware protection mechanisms in recent CPUs (§2.2).

2.1 Memory virtualization approaches

Current VMs use one of the three following mechanisms:

Shadow paging uses hypervisor-managed *shadow page tables*, shown in Fig. 1b, that directly translate a guest virtual addresses (GVA) to host physical addresses (HPA). The guest maintains its own page tables, but the hypervisor forces the MMU to use shadow page tables for address translation. Shadow paging thus offers native translation performance with a one-dimensional page walk.

The hypervisor typically write-protects guest page tables, such that *every guest write* to a guest page table traps into the hypervisor to update the shadow page table [4]. Modern implementations thus need to trap on guest page table writes and on privileged guest instructions, such as TLB flushes. Despite elaborate optimizations [61], shadow paging suffers from these high intrinsic costs for page table manipulation.

The performance of page table manipulation is critical for some workloads, such as function-as-a-service (FaaS). With FaaS, process initialization is on the critical path of function invocations, which includes page table manipulations [25]. To achieve strong isolation, FaaS runtimes are commonly deployed in VMs, e.g., Kata containers [34, 52], which makes page table management a performance-critical operation.

Paravirtualization of MMUs, e.g., in Xen-PV [14], predates hardware virtualization extensions. It can be seen as a variant of shadow paging in which traps are replaced by explicit hypercalls in the guest OS, used to request changes to the hypervisor-managed GPA-to-HPA page tables.

Paravirtualized page tables, however, are costly: hypercall overheads are of the same magnitude as the traps in shadow paging, requiring context switches between VMs and the hypervisor. While paravirtualization can batch modifications to reduce overheads, lazy shadow paging can achieve similar benefits. Therefore, only older hypervisors used paravirtualized page tables by default [12, 14], newer ones use optimized shadow paging and nested paging [13, 66, 73, 75].

Nested paging is a hardware-accelerated approach that performs GVA-to-HPA translation using two hierarchies of page tables: (i) guest (VM-controlled) page tables and (ii) host (hypervisor-controlled) page tables (see Fig. 2). The guest page tables translate GVA-to-GPA (guest physical addresses); the host ones translate GPA-to-HPA. Every GPA in a guest page table requires a GPA-to-HPA translation by the MMU. This procedure is called *two-dimensional page walks* [19].

A two-dimensional page walk multiplies the number of memory accesses per address translation. In the worst case, a single translation must access m levels of the guest page table (horizontal dimension in Fig. 2), where the GPA of each level

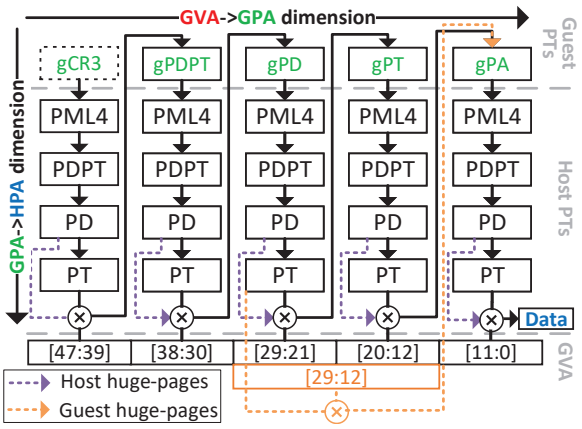


Figure 2: 2-D page table and page walk for nested paging

is first translated by accessing n levels of the host page table (vertical dimension), plus n and m accesses to the contents of the respective page tables: $nm+n+m$ memory accesses in total (e.g., 24 memory access in existing x86–64 processors using 4 KB pages, where $m=n=4$).

Several studies have reported that the overheads of two-dimensional page walks may account for over 30% of application execution time [19, 57]. With the growth in working set sizes and deeper radix page tables [1], this could lead to a quadratic increase in memory virtualization overheads.

Translation overheads can be reduced by using huge pages on the host and/or guest page tables. This bypasses part of the page walk, as shown in the dotted lines in Fig. 2. Their use, however, is not always feasible and may lead to under-utilization of memory due to internal fragmentation [54]. Hardware partial walk caches target similar optimizations but are typically less effective due to their reliance on spatial and temporal reuse [32].

Despite its higher translation costs, nested paging is often the preferred virtualization approach, because it enables guests to perform page table updates without hypervisor intervention while remaining compatible with full virtualization.

2.2 Hardware memory protection

Physical memory protection, recently introduced in commodity CPUs, offers a new hypervisor-controlled mechanism for memory isolation across VMs [60, 64]. AMD SEV-SNP [60] is one example of such technology, which utilizes both memory encryption and physical memory tagging to enhance VM isolation; other architectures, e.g., RISC-V, also offer mechanisms for physical memory protection [71].

In AMD SEV-SNP, the MMU checks each host physical memory access against a *host frame permission table* (called *RMP*) that identifies which VM can access each host frame. The RMP is a physically-contiguous array of memory that contains one entry per host frame. Each entry has a unique identifier of the VM that the host frame is assigned to. Since the MMU checks every HPA against the RMP, this ensures that VMs only access HPAs assigned to them.

RMP checks only happen during a TLB miss, and AMD's implementation has various optimizations to reduce their overhead: (1) RMP entries can be cached as regular data when accessed by the MMU during a page walk, minimizing RMP memory accesses (page table entries can be cached too); and (2) cache lines are extended with their RMP entry to eliminate RMP lookups on cached data.

To decide if it is possible to leverage such hardware memory protection features to accelerate address translation and page table manipulation in VMs, we can consider existing AMD SEV-SNP deployments. SEV-SNP is integrated into Microsoft's Azure cloud platform, and recent performance results show a low overhead for SEV-SNP-enabled VMs [48]. Since SEV-SNP performs both host frame tagging and cache line encryption, with the latter dominating performance overheads [49], using just host frame tagging as part of memory translation should have an even lower overhead (see §5.4).

3 Translation Pass-Through Design

Our design goals and key insights for TPT are as follows:

Native performance. Our solution should offer efficient translation in both current and future systems, where we expect existing memory virtualization approaches to not scale (see §2). Our insight is that, unlike the quadratic overhead of nested paging, VM translation with host physical memory tagging adds a single access to the tag for each page table level. Prior work has shown that such overheads can be largely hidden at the micro-architectural level [6, 7, 16] (see §5.4), and existing commercial results seem to indicate the same [48]. Thus, we make a choice to use tagged physical memory to achieve native translation performance in VMs.

Compatibility with hypervisors/guests. To facilitate adoption, our solution should avoid major changes to existing hypervisors and guest OSs. Achieving this is challenging, as memory translation is deeply ingrained in hypervisor and guest OS implementations. Paravirtualization is often used in virtualized environments in which full hardware virtualization is too complex to implement or too expensive [30, 50, 59]. Nevertheless, a fully paravirtualized memory management interface, such as Xen-PV [74], would require extensive changes to the guest OS, including the boot sequence, I/O layer, and kernel memory management.

To sidestep this complexity, our observation is that TPT's translation approach can be confined to user-space applications, which experience the highest gains. It can be enabled dynamically for each guest process at runtime. As we show in §5.2, TPT is not expected to benefit kernel performance. By limiting TPT use to user-space, we avoid changes to I/O management, guest system boot or guest memory management, and maintain compatibility with existing hypervisor interfaces and host memory management features, such as VM migration or host frame reclamation.

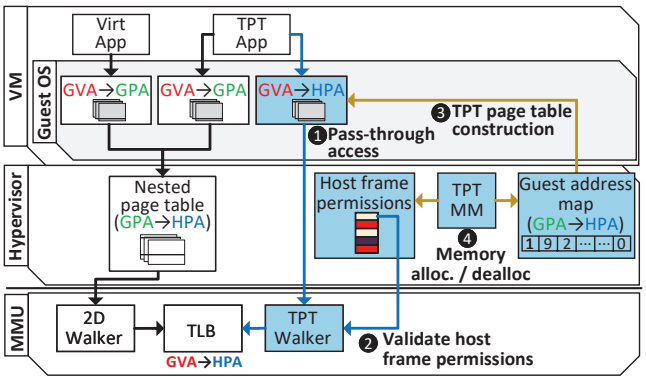


Figure 3: TPT prototype design (Extensions to hardware, hypervisor, and guest are shaded. Memory fast-path translation are blue arrows. Page table management are yellow arrows.)

3.1 Design overview

Fig. 3 shows the main components of our TPT prototype. As a starting point, we assume that the guest OS and hypervisor use nested paging by default; however, our design is general and also applicable to shadow paging (e.g., Linux supports both) or a hybrid system [27, 50]. Further, we assume the availability of hardware memory protection using host frame tags as used internally in AMD SEV-SNP (see §2).

By default, every guest process has a single non-TPT page table (as usual; see "Virt App" in Fig. 3), until the TPT prototype enables TPT on that process. At this point, the guest OS constructs and maintains *dual page tables* for that process, by keeping both a TPT and non-TPT page table in sync (see "TPT App"). We could instead have one or the other depending on whether we enabled TPT on each process, but maintaining dual page tables is inexpensive (evaluated in §6.3), keeps change complexity low, and makes host frame revocation simple to implement: e.g., during host frame reclamation or VM migration, the hypervisor can force all processes to use their non-TPT page table until the guest OS has "repaired" the corresponding TPT page tables (see § 3.2, 3.4 and 4.2).

The guest OS uses the non-TPT page table as the canonical representation of address translation. It only maintains a TPT page table for user-mode execution of processes for which it explicitly requested TPT (step ❶). When running with TPT, the guest updates both page tables, thus keeping them in sync.

To populate the TPT page table, the guest OS retrieves GPA-to-HPA translations from the "guest address map" (step ❸). The guest address map is a data structure used by a VM to know the mapping between its GPAs and the corresponding HPAs. The hypervisor maintains one guest address map for each VM, maps it as a read-only guest physical memory range when a VM boots, and updates it each time the hypervisor changes a guest-to-host physical memory assignment for that VM (step ❹). This approach minimizes changes to the hypervisor since we can reuse GPA faults and existing balloon drivers to manage host frames, and to update the corresponding guest address map and host frame permission table.

The design of dual page tables assumes that each hardware thread has registers pointing to both page tables (TPT and non-TPT), and that the hypervisor may force a VM to use its non-TPT page tables despite also having TPT page tables. We describe the necessary hardware extensions in §4.

3.2 Dual page tables in the guest OS

Every hardware thread has hardware support to access the dual page tables, by using separate page table pointer registers. This ensures compatibility: non-TPT VMs require no changes at all, and TPT-enabled VMs boot without changes — i.e., no TPT page-table is used. A TPT VM then dynamically enables TPT by providing a TPT page table, and can also disable it.

The guest OS enables TPT on a per-process basis, and for user-level code only, where TPT is most effective (see §5.2). TPT can be activated for a process based on an explicit user request. One could extend this with automated runtime policies, although this is out of the scope of this paper; e.g., by monitoring performance metrics such as TLB-miss rates, RSS values, and page-walk cycles.

The guest OS always operates on the canonical non-TPT page table, with its GVA-to-GPA translations, to avoid changes on existing components such as memory management abstractions and algorithms, e.g., to perform reverse virtual address lookups. Each time a non-TPT page table is modified, the guest OS efficiently reflects the changes to the corresponding TPT page table, if any. Entries in a TPT page table take the canonical GPA and translate it to the corresponding HPA using the added guest address map in step ❸, which provides "GPA→HPA" translations specific to this VM.

3.3 Page walks and host frame permissions

The hypervisor assigns a unique identifier to each VM, which is used in the host frame permissions table to mark which host frames are assigned to each VM. When the MMU traverses a TPT page table, it raises an exception into the hypervisor whenever the tag for the VM does not match that of an accessed host frame.

Note that a 4-level page walk in TPT incurs up to 9 memory accesses, but allows micro-architectural optimizations to hide permission checks (see §5.4). If we instead look at a 5-level page table, nested paging goes from 24 to 35 memory accesses, but TPT only goes from 9 to 11 memory accesses, highlighting the advantage of TPT when moving to upcoming architectures with larger physical memory spaces [1].

The behavior of a non-TPT page table is unchanged (see "GVA→GPA" in Fig. 3): a TLB miss triggers a traversal from the MMU, which performs a two-dimensional traversal when using a nested page table set by the hypervisor.

3.4 Host frame management in the hypervisor

The hypervisor tracks GPA→HPA assignments as usual: guest accesses to an unassigned guest frame trigger allocation and mapping into a host frame, i.e., via guest access to an unmapped page in the EPT.

These host frame assignments are captured by the hypervisor's TPT *memory manager* ("TPT-MM" in step ❹). It then updates the host frame permissions used by the MMU in step ❷ and the per-VM guest address map used by the guest OS in step ❸ (see §4.2 for more details).

The hypervisor reclaims host frames from a VM by using the existing balloon driver. When frames are released to the hypervisor, the latter updates the guest address map and host frame permissions accordingly, followed by the invalidation of the TLB entries using existing mechanisms – nested paging uses instruction `INVEPT` in x86–64, whereas shadow paging uses a reverse map to invalidate individual pages.

In some cases, the hypervisor must forcibly reclaim host frames without guest OS cooperation (e.g., the VM is uncooperative or its balloon driver is slow to respond). We design a protocol between the guest OS and hypervisor to handle this case: (1) the hypervisor forcibly disables the use of TPT page tables on that VM and injects a "TPT status" exception onto it. Since the guest has dual page tables, the processor will exclusively use the non-TPT page tables; (2) the hypervisor reclaims any host frames it needs from the VM as usual, removes them from the guest address map, and resets the host frame permissions; (3) the hypervisor resumes guest OS execution; (4) the guest OS gets the injected interrupt and "repairs" the affected page tables to ensure that they do not use the reclaimed host frames; and (5) the guest OS issues a hypercall to notify the hypervisor it can re-enable TPT.

VM migration is handled similarly. The hypervisor disables TPT during migration and notifies the guest OS upon completion by injecting the "TPT status" exception. At this point, the guest OS repairs its TPT page tables based on the new guest address map contents, and re-enables TPT.

Note that this protocol is only needed for TPT-enabled guest processes, which we expect to be a small fraction of all VMs and guest processes. It also only triggered in already expensive cases, such as forceful host frame reclamation, and VM migration. Failure to unmap reclaimed host frames is not a security issue: the VM does not have access to such frames through the host frame permissions, and guest access to an unassigned frame results in an exception in the hypervisor, which can allocate a new frame or terminate the VM.

3.5 I/O host frames and pass-through devices

The host frame permission table only covers the system's main memory address range, which prevents support for pass-through devices on TPT processes (e.g., a DPDK application with VM device pass-through [2, 24]).

To support such additional physical memory address ranges, TPT includes new privileged address range registers, which are configured by the hypervisor to grant a VM access to the selected ranges (selected during VM boot). These ranges are assigned to the executing VM, and the hypervisor exposes their HPAs through the guest address map. This mechanism operates similarly to x86 MTRRs [32] and AMD's IORRs [8].

Name	Description
TPT-cr3	Root TPT page table location (zero disables TPT)
cpuid leaf	Discovery for TPT support
tag_{base,end}	MSR w/ physical addr. of host frame permissions table
VMCS TPT-tag	Tag associated with VM (zero disables TPT)
VMCS TPT-IORR	Address range regs. for VM permissions to HPA ranges
TPT-fault	Exception for invalid permission access
TPT_enable	Hypercall to request TPT enable
TPT_status	Hypervisor-injected int. to signal TPT status change
TPT_addrmap	Virtual PCIe device with guest address map

Table 1: TPT interface added to CPU (Guest ISA (top); hypervisor ISA (middle); guest/hypervisor interface (bottom).)

4 Implementation

We implement a prototype of TPT for Linux 5.16 that consists of 1,700 lines of code (LoC) for the guest OS extensions, 500 LoC in the KVM hypervisor, and 700 LoC in QEMU to configure and start VMs (counted using CLOC [21]). Our prototype targets x86-64, but most changes are architecture-agnostic. Table 1 summarizes the changes visible at the ISA and guest/hypervisor interface, where VMCS identifies the VM hardware configuration fields. In particular, dual page tables are implemented by setting both cr3 and the new TPT-cr3 (which can be disabled by the hypervisor by setting VMCS field TPT-tag to zero).

4.1 Hypervisor extensions

Our hypervisor is based on Linux KVM/QEMU and supports shadow and nested paging by default.

Host frame permissions. The host frame permission table is located in contiguous host physical memory, spanning as many entries as frames in the host physical memory range. With 32-bit tags (already used by AMD SEV-SNP [60]), that corresponds to a 0.1% memory overhead.

The table is configured by the hypervisor, which sets registers tag_{base,end} (similar to AMD RMP_{BASE,END} [8]). In addition, the hypervisor provides a unique TPT identifier for each VM in VMCS field TPT-tag (checked against host frame permission table entries), or sets it to zero to disable TPT (e.g., when migrating a VM or forcing page reclamation).

Extra memory ranges are permissioned by the hypervisor via VMCS TPT-IORR (e.g., for user-level device passthrough; see §3.5), which are used when the requested address is outside the DRAM's physical address space.

Guest address map. The hypervisor generates a mapping for every VM to translate the VM's GPAs to their corresponding HPAs. Upon booting the VM, the hypervisor constructs the guest address and maps it as a read-only guest physical memory range in the VM. Each map is an array in the host virtual memory that covers the guest physical memory range and extra pass-through device ranges assigned to the VM (configured via QEMU). This map is exposed as a virtual PCIe device to VMs (TPT_addrmap), and the size of each map entry is 8 B per 4 KiB frame, resulting in 0.2% memory overhead.

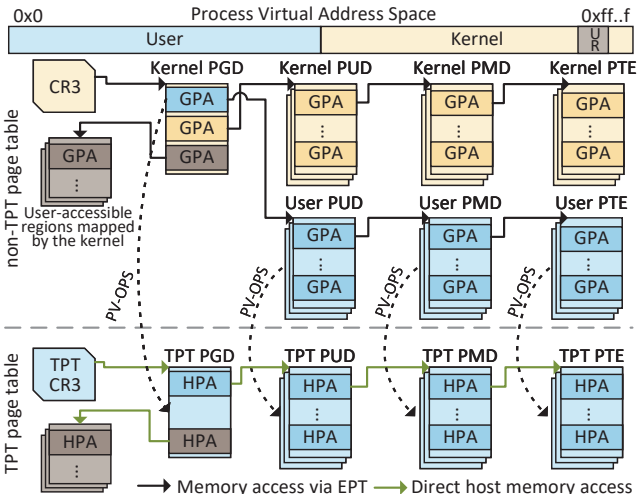


Figure 4: Dual page tables in guest OS (Changes in non-TPT page tables are synchronized to TPT page tables via pv_ops.)

We extend KVM's tdpmmu [63] to update the guest address map's contents when modifying GPA mappings without allocating hypervisor memory on unmapped guest sub-ranges.

4.2 Guest OS extensions

The guest OS changes are largely restricted to a new TPT-specific paravirtualization backend. The new features are activated when the guest OS kernel detects TPT support by the hypervisor (via a new cpuid leaf, configured by the hypervisor). After the discovery of TPT, the guest OS maps the TPT_addrmap device as a write-back (cacheable) memory range as its guest address map.[1]

To enable TPT for a process, the guest user writes to a new procfs entry, which triggers the construction of dual page tables. After the TPT page table has been created, the kernel puts the TPT page table into the new TPT-cr3 register to activate it when a process is rescheduled.

Dual page tables. Our guest OS maintains dual page tables, shown in Fig. 4, and the TPT page tables only cover addresses accessible in user-mode. The guest disables TPT every time it enters kernel-mode and re-enables it when exiting back into user-mode by writing into TPT-cr3 (similar to existing PTI logic [58]). Note that the regular cr3 register always points to the non-TPT page table, in case the hypervisor forcibly disables TPT (see §3.4 and §4.3).

To synchronize the dual page tables, the guest patches the page table operations via a new pv_ops backend when it detects TPT support. pv_ops is an existing Linux kernel API that abstracts core kernel operations in a guest OS to work optimally across different hypervisors, and is used by default on

[1] We modify the kernel's iomap to support cacheable accesses with the right PAT [56] memory attributes. Note that the virtual device's contents are backed by host memory.

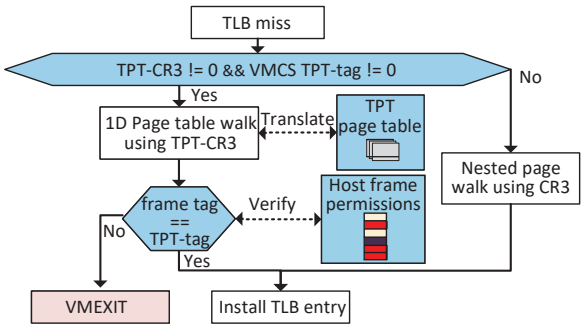


Figure 5: MMU logic to select TPT/non-TPT translation

most VMs. These core operations include page table manipulations, and the overhead of enabling this API is negligible – Linux leverages dynamic code patching to optimize calls to the hypervisor backend selected at boot time.

Every page table modification on a vanilla Linux guest uses GVAs and GPAs and goes through the `pv_ops` mechanism. TPT's `pv_ops` backend performs all requested changes as usual on the non-TPT page tables, but also synchronizes changes to user-accessible addresses with the corresponding TPT page table. Maintenance of dual page tables is therefore transparent to the guest kernel, and requires no more than 3 additional memory accesses to update the TPT page tables. First, the necessary GPA-to-HPA translation is retrieved from the guest address map. Next, the TPT page table entry is located by accessing the extended `mapping` field in `struct page` of the non-TPT page table. Finally, the TPT page table entry is updated with its new HPA value.

In most cases, the guest OS accesses a GPA before mapping it into a page table (e.g., when zeroing it), ensuring that a translation is available in its guest address map. In the few cases in which a GPA is not allocated to a HPA, the guest touches the page to force its presence, going through the pre-existing guest physical memory page-in logic of the hypervisor.

Huge page support. TPT supports huge page translation optimizations, which are used if a page is huge on both the guest and the host (note that the same happens with nested and shadow paging). Our backend adds 12 `pv_ops` functions to support TPT with huge pages.

4.3 MMU extensions

Fig. 5 shows how the hardware extensions for TPT work on a TLB miss. If `TPT-cr3` is supplied by the guest OS and the VMCS `TPT-tag` has not been zero-ed by the hypervisor (disabling TPT), the MMU uses the TPT approach for translation; otherwise, it falls back to using the VM's regular page table walk, e.g., via nested paging, as shown in the figure.

With TPT, the MMU takes each address in the page table from `TPT-cr3` as a HPA. The MMU obtains the HPA's assigned tag from the host frame permission table. It extracts the frame number from the HPA and uses it to index into the permission table. If the retrieved value matches the VMCS field `TPT-tag` (cached in an internal register), the MMU continues to the next page table level; otherwise, it raises a `TPT-fault` exception in the hypervisor. If the HPA falls into any of the VMCS `TPT-IORR` ranges, the MMU considers the HPA valid before accessing the host frame permission table.

Note that various micro-architectural optimizations are possible to perform host frame permission checks during TPT page table traversals, which are discussed in §5.

5 Discussion

Next, we discuss design alternatives, the limitations of the design, and how different design choices relate to them.

5.1 TPT with Xen-PV

TPT is not based on Xen-PV [74] due to performance and compatibility considerations.

To update guest page tables, Xen-PV employs a mechanism called direct-paging that exposes the GPA-to-HPA mappings per guest, similar to TPT's guest address map. However, unlike the TPT model, Xen-PV guests must perform costly hypercalls to update their own page tables. Xen-PV also requires all guest code to execute in ring-3, which introduces hypercall and VM traps to execute privileged guest instructions. Furthermore, Xen-PV exposes a *machine-wide* HPA-to-GPA mapping to *all of its guests*, which is required for page table management operations, and thus reduces inter-VM isolation.

From a compatibility perspective, TPT's KVM-based design is non-disruptive and allows gradual adoption: TPT's hypervisor supports both TPT and non-TPT guests, and TPT guests can run on non-TPT hypervisors (without TPT's benefits). TPT supports full hardware-based virtualization, and provides a modular and adaptable implementation.

5.2 Impact of TPT on kernel-mode

Our guest OS prototype uses TPT page tables during user-mode execution only. This is because Linux kernel-mode accesses have very small translation overheads: it maps all kernel memory using 1 GiB pages [28], making TPT's benefits in kernel mode marginal. In addition, kernel-mode TPT support would be more complex and intrusive: e.g., kernel mode has a linear map for the entire physical address space, and handles physical addresses, such as DMA and contiguous memory allocator (CMA), making the addition of two physical addressing modes (GPA and HPA) more cumbersome. We leave the exploration of kernel-mode TPT to future work.

We quantify the potential impact of kernel-mode support for TPT by measuring the performance of randomly accessing 100 GiB of memory in kernel-mode. We compare shadow paging (with direct GVA-to-HPA translations) and nested paging (with 1 GiB guest kernel pages) and confirm that TPT would provide limited benefits: nested is only 9% and 3% slower than shadow paging when using typical 4 KiB and 2 MiB host pages, respectively, whereas we see overheads of 1.5×–2.5× with the same random access pattern in a user-space application (which does not use 1 GiB pages).

5.3 Impact of TPT on memory de-duplication

By using a single, per-VM tag, TPT cannot support memory de-duplication across TPT processes on different VMs (e.g., via Linux KSM [9]). The same problem exists in AMD SEV-SNP, but is less severe in TPT because only TPT-enabled processes are subject to this limitation: all other processes and kernel-mode pages can still benefit from KSM. It would also be possible to change the hardware to assign multiple `TPT-tag` values to a single VM, and use tag mismatch exceptions to soft-multiplex a larger number.

5.4 Host frame permission check performance

TPT uses per-VM tags to check host frame permissions, something that is inspired by AMD's SEV-SNP [60]. TPT's hardware extensions are feasible with minor changes to AMD's SEV-SNP (described §4.3). Similar changes can be applied to other existing and upcoming architectures that support efficient host memory tag checks such as CHERI [72], RISC-V with PMP [71], and Arm [10, 33].

A naive MMU implementation would perform page table walks and host frame permission checks in sequence, issuing up to 9 memory accesses – a 2.7× improvement over the 24 accesses of nested paging. In practice, there are several micro-architectural optimizations to hide tag accesses and checks: (i) partial walk caches will skip intermediate host frame permission checks; (ii) tags can be embedded into the data they describe to avoid accesses; (iii) tags can be cached to reduce access times; and (iv) host frame permission accesses and checks can be overlapped with page table traversal. More specifically, AMD SEV-SNP embeds host frame tags into the data they tag within the cache hierarchy (reducing the number of accesses), and caches the tag table's contents in the regular cache hierarchy (reducing access latency). Note that some MMU implementations already use the L2$ to cache page table entries, and others have evaluated using a separate tag cache [33]. An optimized MMU implementation would hide permission check accesses by executing them in parallel to page table traversal, which can continue speculatively.

5.5 Security considerations

A malicious or faulty guest in TPT could produce page tables in which any of their levels point to an HPA not assigned to the executing VM (defined as an "incorrect HPA" from here on). An instruction that accesses memory through an incorrect HPA is never committed, but speculatively executing page table walks and permission checks in parallel could lead to potential side-channel attacks, where the page walker logic is used to prime cache lines not assigned to the executing VM.

Single-VM case. We can terminate any such VM that accesses incorrect HPAs, thus avoiding data leakage within a multi-core VM (e.g., to prime/probe cache lines separately).

Inter-VM case. Leakage could happen across colluding VMs: one VM may use the page walker to prime a cache line based on confidential data, and the other VM uses a prime/probe side-channel attack based on the line primed by the first VM [43, 78]. This is a super-set of the single-VM case above.

Such inter-VM side-channels already exist in current systems, since the micro-architectural mechanisms are the same. We can use VM termination, together with existing mitigations to resolve them, and therefore enable aggressive host frame permission check implementations: (1) immediate VM termination ensures that a channel has minimal bandwidth, and an operator can throttle VM creation when frame tag mismatch exceptions rise; (2) integrating VM tags into the cache hierarchy, as done by SEV-SNP, links permission checks and cache accesses, reducing the window of vulnerability to a single memory access (every page table walk access is tag-checked when the cache line is loaded); and (3) existing mitigation techniques are applicable to TPT, such as cache partitioning [42], or controlling the flow of micro-architectural information during speculative execution [36, 76].

Note that the guest address map exposes HPAs set by the hypervisor in response to memory usage of all VMs. This could be used as a side or covert channel between VMs, but the same is valid for memory ballooning or guest physical memory paging. The same, existing mitigations should be applied in all three cases, such as event frequency modulation.

6 Evaluation

Our evaluation demonstrates the performance gains of TPT over traditional virtualization mechanisms. To evaluate our TPT prototype, we conduct a functional emulation of its hardware capabilities on a commodity x86–64 machine, and assume that VMs only map and access their assigned pages.

Our evaluation platform cannot enforce frame tag checks in hardware, and we instead model the performance impact of the frame tag check hardware for two extreme points in the micro-architectural implementation space of the MMU (see *TPT-opt* and *TPT-naive* below). Using this approach, we assess the end-to-end impact of the proposed approach on large-scale workloads, since traditional CPU simulators would make their evaluation unfeasible. While AMD SEV-SNP already implements frame tag checks, we were unable to repurpose it to more directly evaluate TPT; this is because frame tag checks are coupled with nested paging and memory encryption, both of which introduce substantial overheads that we could not isolate.

6.1 Experimental methodology

Testbed. We use a server with 2× Intel Xeon Silver 4216 CPU and 512 GiB of memory (2× 256 GiB DDR4 2,933 GHz). It has an SR-IOV capable NIC (Mellanox ConnectX-4 Lx), which exposes dedicated virtual functions (VFs) for the host and VMs. Intel virtualization support (`VT-x`) and Intel virtualization for direct I/O access (`VT-d`) are enabled for VMs to have direct access to their dedicated VFs (using `vfio-pci` pass-through). Hyperthreading is disabled, the frequency governor is set to "performance", and "turbo" is disabled for

stable results. Both VMs and hypervisor run Ubuntu 20.04 with Linux kernel 5.16. VMs are managed by QEMU with KVM acceleration, have 16 vCPUs with 156 GiB of memory, and each vCPU is pinned to a separate physical core.

We use NUMA node 0 to evaluate both native and virtualized executions. NUMA node 1 executes client agents (without virtualization) for workloads that require requests over the network, which are passed to the physical NIC, and routed via the NIC's internal switch to the correct VF.

Hardware emulation and performance modeling. We emulate the proposed hardware extensions for TPT on the x86-64 platform in two ways:

(1) Host frame tag checks. Our base results execute on the commodity machine "as-is" and assume an optimized MMU implementation where host frame permission checks have no performance impact (*TPT-opt* below). This is a reasonable approximation as SEV-SNP has micro-architectural optimizations to hide the latency of host frame permission accesses (see §5.4), whereas traversal and check operations can be executed in parallel without compromising security (see §5.5).

We also model the performance of a naive MMU implementation where traversals and checks execute sequentially (*TPT-naive* below), resulting in an extra memory access on every page-walk cache miss. We obtain a conservative performance estimate of the naive MMU implementation by placing TPT page tables on a different NUMA node from the one with executing cores; this effectively doubles the access latency, from 81 ns to 161 ns, respectively, according to MLC [69]. A similar technique was used to model larger latencies in prior works [40,77]. This is a reasonable approximation on existing hardware since SEV-SNP avoids tag accesses by extending cache lines, and hides tag accesses by caching them (others have also proposed separate caches to avoid capacity conflicts [33]). Note that we cannot model the added memory bandwidth consumed by tag accesses, but other tagged systems show overheads below 2% on most applications, and as low as 8% in the worst case [33].

(2) MMU walker logic. Current platforms lack the necessary hardware for registers `cr3` and `TPT-cr3` and the additional MMU logic we propose to manage them, as shown in Fig. 5. We therefore emulate these extensions in software; we modify the hypervisor to intercept `cr3` operations in TPT processes, select between `cr3` or `TPT-cr3`, and enable EPT or TPT translation modes, respectively.

We add support in KVM for per-vCPU EPT control, and patch the guest OS PTI [58] assembly thunks to perform the following hypercalls when executing TPT processes:

(i) Kernel-to-user: disable EPT on the vCPU, set the guest's `cr3` to `TPT-cr3`, intercept and emulate guest `cr3` reads to return the guest's original `cr3` value (sometimes performed in exception handling during guest execution).

(ii) User-to-kernel: enable EPT on the vCPU, restore the guest's original `cr3` value, and disable `cr3` read interception.

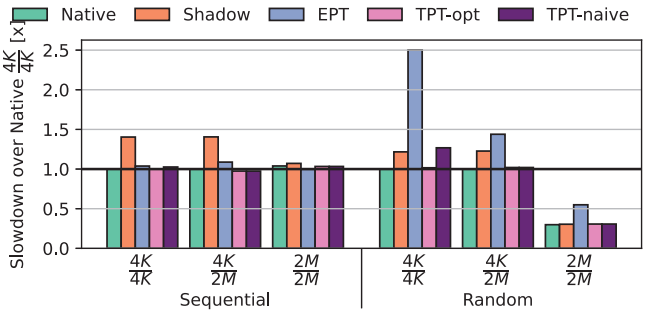


Figure 6: Relative slowdown for different translation mechanisms over native $\frac{4K}{4K}$ in memory access micro-benchmark (Lower is better.)

With a hardware implementation, the decision to use `cr3` or `TPT-cr3` would be performed by hardware with negligible cost. However, our software emulation has additional overheads for performing the additional hypercalls (via `VMCALL` and `VMEXIT`), which may dominate execution time on system call- or interrupt-heavy workloads (PTI is also only enabled for TPT).

We therefore report execution times after subtracting the software emulation overheads (time spent on new hypercalls performing EPT and `cr3` manipulations) from the application execution time. Note that most benchmarks do not invoke the hypercalls during the evaluation phase, and thus we do not remove the emulation overhead in such cases. For experiments that need the emulation hypercalls, such as the page table manipulation micro-benchmarks, we configure them to execute in a single physical CPU to maintain modeling correctness.

Configurations. We evaluate TPT's performance against the following system configurations:

(a) Native: Native execution, which serves as our ideal, upper bound on performance.

(b) Shadow: VM with shadow paging.

(c) EPT: VM with nested paging using Intel's extended page tables (EPT) mechanism.

(d) TPT-opt: VM with an optimized TPT implementation.

(e) TPT-naive: VM with a naive TPT implementation.

We evaluate each of the configurations under different host/guest page sizes: (1) $\frac{4K}{4K}$: guest OS uses base pages (4 KiB), and host backs VM with base pages (4 KiB); (2) $\frac{4K}{2M}$: guest OS uses base pages (4 KiB) and host backs VM with huge pages (2 MiB); and (3) $\frac{2M}{2M}$: guest uses huge pages (2 MiB) by enabling transparent huge pages (THP), and host backs VM with huge pages (2 MiB).

6.2 Memory translation

We create a single-thread memory micro-benchmark to assess TPT's impact on memory performance under different scenarios. Our benchmark allocates a 100 GiB buffer to serve as the target for memory read operations at a 64-bit granularity.

First, we evaluate the performance of sequential and ran-

Shadow	EPT	TPT-opt	TPT-naive
18.8×	5×	6.68×	6.68×

Table 2: Relative slowdown for different translation mechanisms over native $\frac{4K}{4K}$ when manipulating the guest page table using `mprotect` (Lower is better.)

dom memory accesses. Fig. 6 shows the relative slowdown of the evaluated system configurations over the native execution time. TPT-opt's performance matches that of Native, regardless of the memory page size, under both sequential and random access patterns. This is expected, as TPT-opt eliminates all virtualization-induced memory translation overheads due to its use of direct GVA-to-HPA page tables.

We also observe that EPT has the worst performance under the random access pattern due to its two-dimensional page walk on each TLB miss. Huge pages reduce, but do not eliminate the overhead of EPT, as the page walk is shortened due to the smaller size of the two-dimensional page walk [47].

TPT-naive only underperforms in random memory accesses under the $\frac{4K}{4K}$ configuration, where it exhibits a 1.25× relative slowdown over TPT-opt and Native. This is the result of the longer page walk duration on TLB misses, but it is still 2× faster than EPT under the same configuration.

Shadow exhibits a slowdown of 1.41× and 1.25× over Native for sequential and random memory accesses, respectively. The overheads stem from the extra time spent in the hypervisor due to page faults, which cause expensive VMEXITs. Although the guest OS page tables are populated, the shadow page tables are maintained by the host and updated lazily and on-demand: accesses to newly-mapped pages incur a page fault that is handled by the hypervisor, which in turn updates the shadow page tables and resumes guest execution.

Conclusions: TPT-opt exhibits native performance, outperforming both Shadow and EPT.

6.3 Page table management overheads

Raw page table manipulation. We evaluate the performance of page table modification under all configurations, as each incurs overheads from different sources. We measure the time taken to downgrade a single page from read-write to read-only via the `mprotect` system call.

Table 2 shows the results, normalized to Native execution. Shadow exhibits a slowdown of more than 18× over Native, as downgrading permissions in the page tables require TLB invalidations that are trapped by the host to amend the shadow page tables. EPT does not require interventions by the host and incurs a slowdown of 5× for a single page permission modification. This is the result of a single TLB entry invalidation in the guest (using instruction `INVLPG`), which invalidates *all* paging-structure translation caches of the current context, including the partial-walk caches (PWC) [8, 32, 47]. We corroborate this finding by observing an increase in the number of cycles the hardware page table walkers are active under the

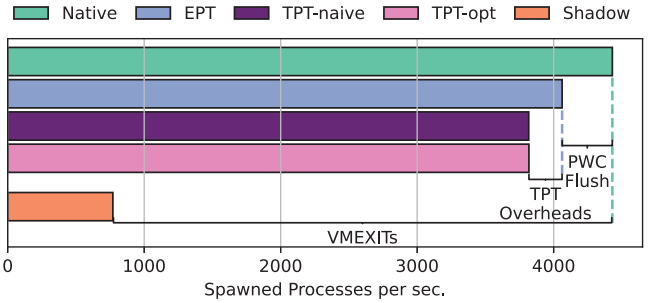


Figure 7: Spawn micro-benchmark for different translation mechanisms (Higher is better.)

Workload	Description	RSS
kcbench	Kernel compilation benchmark (v4.19) [35]	1 GiB
XSBench	Monte Carlo neutron transport algorithm [65]	99 GiB
Canneal	Optimization for chip design (PARSEC [20])	109 GiB
GUPS	Random integer updates in memory (HPCC [44])	129 GiB
PR	Page Rank (GAPBS [17]) on kron graph[2]	72 GiB
BFS	BFS Algorithm (GAPBS) on kron graph	70 GiB
CC	CC Algorithm (GAPBS) on kron graph	70 GiB
Memcached	Facebook ETC [11] (3×10^8 keys; mut. client [39])	108 GiB

Table 3: Application workloads and memory footprint

EPT configuration over Native (not shown).

TPT-opt exhibits a small overhead over EPT due to the additional operations to keep dual page tables in sync. Note that TPT-opt is also subject to the same cache invalidation overheads triggered by `INVLPG`, and TPT-naive has no additional overheads because page table contents are always accessed via non-TPT page tables in kernel space.

We perform the same experiment to measure the performance of mapping anonymous memory, by evaluating the `mmap` system call with the `MAP_POPULATE` flag. We do not observe a performance difference between the configurations, as the majority of time is spent on physical memory allocations and zeroing page contents.

TPT's extra logic to manipulate dual page tables has a small performance impact, and does not affect the end-to-end results on our evaluated applications (see §6.4). Such low overheads to synchronize modifications across page tables are corroborated by prior work [3, 53].

Process spawning. We evaluate the performance of Spawn from the Unixbench benchmark suite [67]. Spawn is a fork/wait-type workload, which measures the number of times a process can fork and reap a child that immediately exits.

Fig. 7 shows that EPT, TPT-opt, and TPT-naive are subjected to the adverse effects of PWC flushes as the `fork` system call performs TLB invalidations. The additional operations in TPT (maintaining dual page tables) lead to a 1.06× slowdown over EPT. In comparison, shadow incurs a 5.18× slowdown over EPT due to `VMEXIT`s induced by TLB invalidation and page faults.

[2] Kron graph [38] scale: 2^{29}, average degree: 16

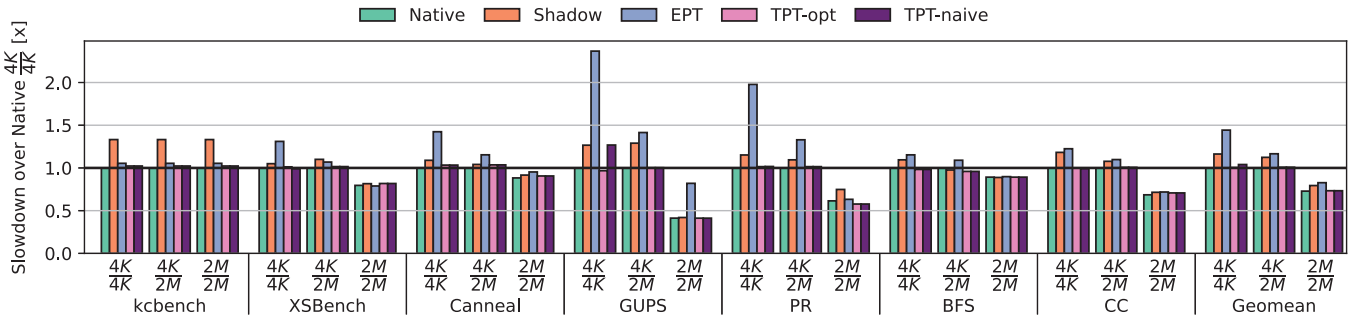


Figure 8: Relative slowdown on applications benchmarks over native $\frac{4K}{4K}$ (Lower is better.)

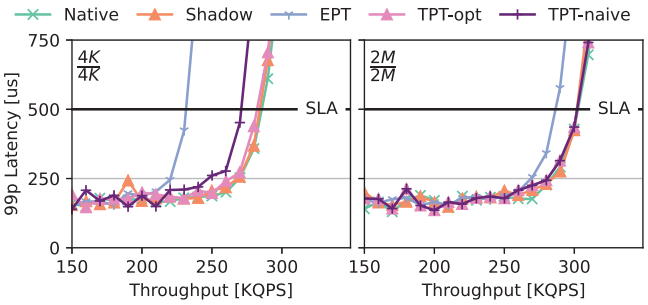


Figure 9: Memcached throughput-latency for different translation mechanisms (Lower is better.)

Conclusions: TPT enables substantially faster page table manipulations compared to Shadow by eliminating `VMEXITs`. It exhibits higher page table manipulation costs over EPT and native, but they have negligible impact on end-to-end application performance, as shown in §6.4.

6.4 Application benchmarks

We evaluate TPT on the application benchmarks listed in Table 3, which are commonly used to evaluate of data centers workloads [27, 53]. All benchmarks, apart from Memcached and kcbench, execute with 8 threads. We execute Memcached with a single thread because we do not have enough client cores to saturate more server threads. For maximum performance, Memcached uses the VMA [41] library for user-level I/O. We evaluate kcbench with a single thread because we need to model performance in the face of emulation hypercalls (as explained in §6.1).

Fig. 8 shows the relative slowdown of all the evaluated configurations over Native $\frac{4K}{4K}$. TPT-opt matches the performance of Native for all workloads, under all page size configurations. EPT exhibits significant slowdowns in workloads with random memory accesses, such as GUPS and PR. Overheads for Shadow are apparent in workloads that perform memory mappings, such as process spawning (kcbench), and dynamic memory allocation with page table modifications (CC and GUPS). TPT-naive exhibits a slowdown only on GUPS with the $\frac{4K}{4K}$ configuration, of 1.25×. GUPS is a random memory access benchmark, which correlates with our previous results for the random access micro-benchmark in Fig. 6. The geometric mean of the slowdown for TPT-naive is of just 3%.

Huge pages reduce the virtualization overheads of both EPT and Shadow, as well as improve the performance of Native execution, because they reduce TLB misses and page walk costs. Huge pages substantially decrease overheads in EPT as they reduce the number of steps on each dimension of the walk. EPT, however, still exhibits noticeable slowdowns with huge pages over TPT-opt and Native. Shadow's overheads with huge pages decrease, as 2 MiB mappings, compared to 4 KiB ones, induce less `VMEXITs` to sync the shadow page tables with the guest's page table mappings.

Memcached. We single out the Memcached benchmark, because it is latency-sensitive. Fig. 9 shows the throughput-latency graph of the 99th percentile of Memcached serving Facebook's ETC requests, with an SLA of 500 µs (following previous work [18]). Although the ETC access distribution is skewed, the keys are small in size and randomly distributed. This affects the overall access distribution of the workload, which exhibits a random memory access pattern.

EPT performs the worst due to the random memory accesses. Shadow, Native, and TPT-opt perform similarly in both page size configurations. This is expected because no new memory allocations occur during the measured portion of the workload. TPT-naive under the $\frac{4K}{4K}$ configuration crosses the SLA with 4% lower throughput than TPT-opt. The mean latency exhibits the same behavior as the 99th percentile, although the knee of the curves occurs at a higher throughput.

Conclusions: The performance of TPT matches Native, and systematically outperforms Shadow and EPT on all benchmarks where page table management or memory access performance dominates, respectively, even with huge pages.

6.5 Impact of 1 GiB huge pages

We now evaluate the same applications in §6.4 with EPT $\frac{2M}{1G}$, using 1 GiB host pages (typically unfeasible in a production system). The applications in Fig. 8 with 1 GiB pages only show a 2.5% speedup (geometric mean) compared to our previous EPT $\frac{2M}{2M}$ results, which would correspond to a 1.16× slowdown over Native and TPT-opt. In turn, Memcached's throughput only increases by 2.5% with 1 GiB pages, which

corresponds to a 1.04× slowdown over TPT-opt and Native.

Conclusions: Unlike TPT, 1GiB huge pages do not eliminate nested paging overheads completely.

6.6 Memory overheads

Guest. Non-TPT processes have no memory overheads, but TPT processes incur a small overhead to hold the TPT page tables. The TPT page tables only map the user space pages of the process and do not map the entire system memory and kernel space. A process with a sequential memory mapping of n pages incurs an additional $(1-\lfloor\frac{\text{page\_size}}{2\,\text{MiB}}\rfloor)\cdot\lceil\frac{n}{2^{36}}\rceil+\lceil\frac{n}{2^{27}}\rceil+\lceil\frac{n}{2^{18}}\rceil+\lceil\frac{n}{2^{9}}\rceil$. For example, a mapping of 1 GiB with 4 KiB pages incurs an extra 2 MiB-worth of TPT page tables.

Host. TPT's guest address map requires 8 B of host memory per GFN to hold the HFN. Therefore, a 64 GiB VM consumes 256 KiB or 128 MiB of host memory if the host utilizes 2 MiB or 4 KiB pages respectively (less than 0.2% in both cases).

Conclusions: TPT only adds small memory overheads in guests and hosts, making it practical for adoption.

7 Related Work

Prior work either attempts to improve existing virtualization mechanisms [27, 31, 46, 50, 53, 70], thus inheriting their shortcomings, or introduces invasive hardware changes, potentially changing the behavior of VMs compared to native execution [5–7, 15, 19, 23, 45, 55, 62].

Hardware-based virtualization. DVMT [6] proposes a software MMU architecture where applications/VMs can use their own address translation structures. DVMT also uses tag-based frame protection for isolation, but retains a two-dimensional translation approach in VMs, albeit with customizable translation structures on each dimension.

Sha et al. [19] propose new paging schemes for processors with software MMUs. The schemes reduce the page walk cost in nested paging by incorporating flat nested page table [5], or reduce the cost of updating guest page tables in shadow paging by intercepting TLB flushes. However, it introduces a software MMU and constraints to guest physical address space size, making it difficult to apply to modern machines and large memory sizes, respectively.

Several studies propose to redesign paging structures. Compromis [23] uses direct segments, but requires the reservation of variable-length physical memory areas for segments, which significantly compromises the flexibility of memory management in hypervisors. Chang et al. [55] propose to flatten 2 levels of page tables to reduce the cost of page walks by half. This approach increases the cost and complexity of managing page tables, and its cost reduction is limited. Nested Elastic Cuckoo Page Tables [62] utilize hashed page tables to reduce the nested page walk cost. However, replacing the existing radix page tables with hashed page tables requires significant changes to existing software and hardware ecosystems.

Caching and prefetching are also effective at hiding translation latency. Thomas et al. [15] explored new MMU caches, including today's partial walk caches in AMD and Intel processors. ASAP [45] reduces address translation latency by storing multiple page tables contiguously and introducing a hardware prefetcher for page walks. Caching does not fully eliminate the memory translation costs, and prefetching can result in mispredictions and numerous memory accesses to the page table when accessing large virtual memory areas. MMU caches and prefetching are directly applicable to TPT.

Improving virtualization. Agile paging [27] combines shadow and nested paging to reduce the hypervisor intervention cost on page table updates. It requires more memory accesses per TLB miss than native machines and TPT.

On-demand virtualization [31] enables virtualization dynamically to migrating bare-metal machines, and disables virtualization after the migration. This approach only applies to bare-metal machine migration, and cannot be generalized: it does not support more than a single VM in a bare-metal machine, and cannot enforce isolation between the VM and the hypervisor, because it relies on identity mappings (1:1) for nested translation between the VM and the hypervisor.

8 Conclusions

TPT is a new approach to memory virtualization, which achieves near-native translation performance for memory-intensive applications in VMs. In TPT, VMs regain control over their translation structures, while maintaining memory isolation across VMs by leveraging emerging physical memory protection technologies. TPT is compatible with both TPT and non-TPT guests, and can be selectively applied to processes running within any TPT-aware VM.

Acknowledgements

We gratefully acknowledge support from the Israel Science Foundation (grants 980/21 and 1027/18). This work was also partially supported by the Technology Innovation Institute (TII) through its Secure Systems Research Center (SSRC), and by JSPS KAKENHI grant number 18KK0310 and JST CREST grant number JPMJCR22M3, Japan.

References

[1] 5-Level Paging and 5-Level EPT. Technical report, Intel Corp., December 2016.

[2] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3), 2006.

[3] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth*

International Conference on Architectural Support for Programming Languages and Operating Systems, pages 283–300, 2020.

[4] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for X86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 2–13, New York, NY, USA, 2006. Association for Computing Machinery.

[5] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 476–487, 2012.

[6] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 457–468, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 515–528. IEEE, 2020.

[8] AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf. Last accessed: December 2022.

[9] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing Memory Density by using KSM. In *Proceedings of the Linux Symposium*, pages 19–28. Citeseer, 2009.

[10] Arm. *Armv8.5-A Memory Tagging Extension*, August 2019.

[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[12] S Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System Performance Evaluation of Para-virtualization, Container Virtualization, and Full Virtualization using Xen, Openvz, and Xenserver. In *2014 Fourth International Conference on Advances in Computing and Communications*, pages 247–250. IEEE, 2014.

[13] S Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *2014 fourth international conference on advances in computing and communications*, pages 247–250. IEEE, 2014.

[14] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[15] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 48–59, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Thomas W Barr, Alan L Cox, and Scott Rixner. SpecTLB: A Mechanism for Speculative Address Translation. *ACM SIGARCH Computer Architecture News*, 39(3):307–318, 2011.

[17] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03619*, 2015.

[18] 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, 2014.

[19] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–35, 2008.

[20] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "the parsec benchmark suite: Characterization and architectural implications". In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[21] CLOC: Count Lines of Code. https://github.com/AlDanial/cloc. Last accessed: December 2022.

[22] Confidential Computing Consortium. https://confidentialcomputing.io/. Last accessed: December 2022.

[23] Boris Teabe Djomgwe, Peterson Yuhala, Alain Tchana, Fabien Hermenier, Daniel Hagimont, and Gilles Muller. (No) Compromis: Paging Virtualization Is Not a Fatality. In *VEE 2021-17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–12, 2021.

[24] Yaozu Dong, Jinquan Dai, Zhiteng Huang, Haibing Guan, Kevin Tian, and Yunhong Jiang. Towards High-Quality I/O Virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–8, 2009.

[25] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[26] Wei Fan and Albert Bifet. Mining Big Data: Current Status, and Forecast to the Future. *ACM SIGKDD Wxplorations Newsletter*, 14(2):1–5, 2013.

[27] Jayneel Gandhi, Mark D Hill, and Michael M Swift. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 707–718. IEEE, 2016.

[28] x86, mm: Enable GBPAGES Option by Default. https://github.com/torvalds/linux/commit/9e899816d126cc6f7d405c349f65363214fe7399. Last accessed: December 2022.

[29] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[30] Irfan Habib. Virtualization with KVM. *Linux Journal*, 2008(166):8, 2008.

[31] Jaeseong Im, Jongyul Kim, Youngjin Kwon, and Seungryoul Maeng. On-demand Virtualization for Post-copy OS Migration in Bare-metal Cloud. *IEEE Transactions on Cloud Computing*, pages 1–1, 2022.

[32] Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 3A: System Programming Guide*, 2022.

[33] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648. IEEE, 2017.

[34] The Operstack Foundation. Kata Containers - The Speed of Containers, the Security of VMs. https://katacontainers.io/. Last accessed: December 2022.

[35] kcbench: Linux Kernel Compilation Benchmark. https://gitlab.com/knurd42/kcbench. Last accessed: December 2022.

[36] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the Spectre of a Meltdown with Leakage-free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[37] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual Remote I/O. *ACM SIGARCH Computer Architecture News*, 44(2):49–65, 2016.

[38] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 133–145. Springer, 2005.

[39] Jacob Leverich. Mutilate: High-Performance Memcached Load Generator, 2014.

[40] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[41] LibVMA. https://github.com/Mellanox/libvma/wiki/Architecture. Last accessed: December 2022.

[42] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418. IEEE, 2016.

[43] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[44] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE*

Conference on Supercomputing, volume 213, pages 1188455–1188677, 2006.

[45] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1023–1036, 2019.

[46] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. Ptemagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–223, 2021.

[47] Timothy Merrifield and H Reza Taheri. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 25–35, 2016.

[48] Microsoft Azure Confidential Computing Powered by 3rd Gen EPYC™ CPUs. https://community.amd.com/t5/business/microsoft-azure-confidential-computing-powered-by-3rd-gen-epyc/ba-p/497796. Last accessed: December 2022.

[49] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of Intel SGX and AMD memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018.

[50] Jun Nakajima, Qian Lin, Sheng Yang, Min Zhu, Shang Gao, Mingyuan Xia, Peijie Yu, Yaozu Dong, Zhengwei Qi, Kai Chen, et al. Optimizing Virtual Machines Using Hybrid Virtualization. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 573–578, 2011.

[51] Neiger, Gil and Santoni, Amy and Leung, Felix and Rodgers, Dion and Uhlig, Rich. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), 2006.

[52] OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com/. Last accessed: December 2022.

[53] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized numa servers with vmitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–210, 2021.

[54] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360, 2019.

[55] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Every Walk's a Hit: Making Page Walks Single-Access Cache Hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–141, 2022.

[56] PAT (Page Attribute Table). https://www.kernel.org/doc/html/latest/x86/pat.html. Last accessed: December 2022.

[57] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html. Last accessed: December 2022.

[59] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[60] AMD SEV-SNP. "strengthening vm isolation with integrity protection and more". *White Paper, January*, 2020.

[61] The x86 KVM Shadow MMU. https://www.kernel.org/doc/Documentation/virtual/kvm/mmu.txt. Last accessed: December 2022.

[62] Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 84–97, New York, NY, USA, 2022. Association for Computing Machinery.

[63] TDP MMU. https://lwn.net/Articles/832835/. Last accessed: December 2022.

[64] Intel Trust Domain Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf. Last accessed: December 2022.

[65] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[66] Ubuntu 14.04 On Amazon EC2: Xen PV vs. HVM. https://www.phoronix.com/review/amazon_ec2_pvhvm. Last accessed: December 2022.

[67] Unixbench. https://github.com/kdlucas/byte-unixbench. Last accessed: December 2022.

[68] Prashant Varanasi and Gernot Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[69] V Viswanathan, Karthik Kumar, and T Willhalm. Intel Memory Latency Checker. *Intel Corporation*, 2013.

[70] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Selective Hardware/Software Memory Virtualization. *ACM SIGPLAN Notices*, 46(7):217–226, 2011.

[71] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture. *RISC-V Foundation*, 2019.

[72] Jonathan Woodruff, Robert NM 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. IEEE, 2014.

[73] Xen Project Software Overview. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. Last accessed: December 2022.

[74] Xen Project. X86 Paravirtualised Memory Management. https://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Management. Last accessed: December 2022.

[75] Xen PV Performance Status and Optimization Opportunities. https://www.slideshare.net/xen_com_mgr/xen-pv-performance-status-and-optimization-opportunities. Last accessed: December 2022.

[76] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[77] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[78] Yuval Yarom and Katrina Falkner. FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX security 14)*, pages 719–732, 2014.

Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management

Yaohui Wang, Ben Luo, Yibin Shen
Alibaba Group

Abstract

In virtualization systems, guest memory overcommitment helps to improve the utilization of the host memory resource. However, the widely adopted I/O passthrough technique makes this task not intuitive since the hypervisor must avoid DMA (Direct Memory Access) failures when the I/O device accesses the guest memory. There already exist several solutions, for example, IOPF (I/O Page Fault) can fix DMA failures by allowing page fault triggered from the I/O device side, vIOMMU and coIOMMU avoid DMA failures by monitoring the DMA buffers in the guest. However, these solutions all face the performance concerns introduced by the memory backup/restore mechanism, *i.e.*, memory swapping. Some free page based methods (*e.g.*, Ballooning, Free Page Reporting, Hyperupcall) are free from memory swapping, but they either are not DMA-safe or introduce high guest communication overhead. In this paper, we propose V-Probe, a high-efficiency approach to achieve memory overcommitment for I/O passthrough enabled VMs. Using fine-grained page meta-data management, V-Probe allows the hypervisor to inspect and reclaim guest free pages actively and efficiently while guaranteeing DMA-safety. Experiments show that, for both memory reclamation and reallocation, the overhead of V-Probe is in the scale of microseconds, which is faster than Ballooning and IOPF based methods by two orders of magnitude. And our micro-benchmark and macro-benchmark show that V-Probe limits the performance impact of memory overcommitment to a low level.

1 Introduction

In virtualization systems, the total memory size of a VM (Virtual Machine) is commonly constant while it is running, but the working set of memory (*i.e.*, memory accessed actively) inside a VM is usually a subset of the total memory [11, 17, 36, 42, 47]. This results in the inefficiency of memory resource utilization [28, 39]. Memory overcommitment [4,6,7,9,19,26,37,40,44] helps to mitigate this problem. It reclaims cold memory (*i.e.*, memory not accessed recently) from a VM, and reallocates the memory to other VMs on demand to increase memory utilization.

However, the widely adopted I/O passthrough technique [3, 5, 12, 15, 26, 27, 41], which significantly reduces the overhead of I/O virtualization [1, 21, 24], requires the hypervisor to keep a fixed memory mapping for the VM during its life cycle to avoid potential DMA failures [4, 26, 40]. This highly limits the ability of memory overcommitment since the memory reclamation will change the memory mapping of a VM dynamically (Section 2).

Previous work tries to solve the contradiction between I/O passthrough and memory overcommitment in two different ways. The first is to introduce the IOPF (I/O Page Fault) mechanism [25, 26, 34]. It allows an I/O device to notify the hypervisor by triggering page faults when the target DMA buffer does not reside in the main memory. After the hypervisor repairs the IOPT (I/O Page Table) entry for the faulted IOVA (I/O Virtual Address), the device can replay the previously failed DMA request. The second is to monitor the guest DMA buffers using the PV (Para-virtualization) technique [4, 10, 11, 40, 44]. Such solutions include vIOMMU [4] and coIOMMU [40]. They use frontend drivers in the guest to inform the hypervisor with DMA buffer alloc/free events. With such knowledge, the hypervisor always keeps the memory mapping of the DMA buffers and thus prevents DMA failures.

Although the above solutions allow memory overcommitment to coexist with the I/O passthrough technique, they face several deficiencies. The first is the compatibility issue, which makes them currently less practical. For example, the IOPF solution requires designated hardware which is not widely supported by hardware manufacturers. And the coIOMMU solution needs changes to the guest OS and is still not supported by off-the-shelf OSes (*e.g.*, the Linux upstream). But even though the compatibility issue can be fixed over time (by evolving the hardware/software), the second issue – performance issue – is unavoidable. When doing memory reclamation/reallocation, these solutions need to backup/restore the

memory content using memory swapping [6, 20, 29, 30, 32]. The overhead it introduces not only slows down the memory reclamation process, but may also cause performance oscillations to the VM and hurts the VM's SLO [6, 49, 50].

While memory swapping is costly, reclaiming guest free pages [44, 45] is a good way to eliminate such overhead, since the content in free pages is meaningless. But because of the semantic gap in virtualization systems, free page reclamation methods usually rely on the communication with the frontend driver running inside the guest to obtain the knowledge of guest free pages. The guest communication introduces extra overhead and increases the response time for the memory reclamation requests (Section 5.1.2). The recently proposed free page inspecting method – Hyperupcall [7] – helps to eliminate such overhead. When doing memory reclamation, Hyperupcall allows the hypervisor to actively invoke the guest injected eBPF functions to inspect guest free pages. However, guaranteeing DMA-safety is a challenging task with this method. In Hyperupcall, since the guest is agnostic to the hypervisor's memory reclamation actions, it may allocate a free page, whose underlying physical page is reclaimed by the hypervisor, as a DMA buffer. Such behavior cannot be perceived by the hypervisor, so it does not have a chance to repair the memory mapping for the DMA buffer, which will further cause DMA failures (Section 2.2.3).

In this paper, we propose V-Probe to address the performance and DMA-safety challenges of memory overcommitment for I/O passthrough enabled VMs. V-Probe targets guest free pages to eliminate the costly overhead of memory swapping. Inspired by Hyperupcall [7], V-Probe uses guest injected helper functions to detect guest free pages actively, which avoids the communication overhead with the guest. But instead of using eBPF, V-Probe uses raw binary helper functions. This avoids the complex eBPF dependencies in the guest OS required by Hyperupcall. Such simplicity makes the deployment of the method easier. V-Probe uses the SFI (Software Fault Isolation) technique [13, 31, 38, 43, 48], which performs strict rule-based instruction-level checks to the injected binary, to prevent malicious code. Using fine-grained page meta-data management, V-Probe not only manages the memory mapping of the reclaimed free pages but also monitors their corresponding page meta-data. This allows V-Probe to react to guest memory allocation events and prevent potential DMA failures. Experiments show that the overhead of V-Probe is in the scale of microseconds. Compared to Ballooning and IOPF, it is faster by two orders of magnitude in both memory reclamation and reallocation. Our micro-benchmark and macro-benchmark show that V-Probe limits the performance impact of memory overcommitment to a low level.

We summarize our contribution as follows:

- We conduct a systematical study of previous VM memory overcommitment methods and characterize these methods in different aspects to provide an overview.
- We propose V-Probe, an efficient memory overcommitment method that targets guest free pages and guarantees DMA-safety for I/O passthrough enabled VMs.
- We evaluate the overhead of V-Probe, and assess its performance in both micro-benchmark and macro-benchmark tests. Results show that V-Probe achieves low overhead and limits the performance impact of memory overcommitment to a low level.

2 Motivation

2.1 I/O Passthrough & DMA-safety

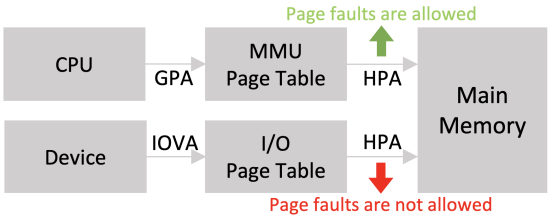


Figure 1: The address translation for CPU and I/O devices in modern hardware architectures. Page faults in the I/O device side will trigger DMA failures.

I/O passthrough, which allows the guest OS to directly interact with the underlying hardware, is widely used in virtualization systems [15, 16]. Using I/O passthrough, a peripheral device can directly access the guest memory through DMA without the intervention of the CPU. It significantly decreases the overhead of guest I/O operations and is also required by the emerging high-performance RDMA (Remote Direct Memory Access) applications [22, 23, 46]. This makes I/O passthrough an irreplaceable part of virtualization systems like the public cloud.

In the same way that the CPU MMU (Memory Management Unit) needs a page table to translate GPA (Guest Physical Address) to HPA (Host Physical Address), devices' DMA requests rely on the IOPT to translate IOVA (I/O Virtual Address, it is usually the same as GPA) to HPA. On the CPU side, if the accessed GPA does not have a valid page table entry, then a page fault will be triggered. The hypervisor will try to fix the memory mapping in the page table, and then resume the guest OS execution from where it triggers the page fault. On the device side, to add IOPF support, the device needs the ability to replay the once failed DMA request after the hypervisor fixes the invalid IOPT entry. But unfortunately, most off-the-shelf devices do not support such mechanism. So they require that all DMA requests must always succeed or they will result in DMA failure and crash of the guest OS (Figure 1). This implies that an I/O passthrough enabled VM has no tolerance to IOPT entry missing [4, 25, 26, 40].

Table 1: The characteristics of the memory reclamation methods from different dimensions.

Methods	DMA-safety	Hardware Compatibility	Guest Compatibility	Main Overhead
IOPF	DMA-safe	Dedicated hardware	No requirements	Memory swapping
vIOMMU	DMA-safe	General hardware	Frontend driver	Memory swapping
coIOMMU	DMA-safe	General hardware	Frontend driver	Memory swapping
Ballooning	DMA-safe	General hardware	Frontend driver	Guest communication
Free Page Reporting	DMA-unsafe	General hardware	Frontend driver	Guest communication
Hyperupcall	DMA-unsafe	General hardware	eBPF tool-chain	Extremely low overhead

In memory overcommitment, the MMU page table entry and the IOPT entry of the reclaimed page will both be invalidated after memory reclamation. As the hypervisor is agnostic to whether a page is used for DMA in the guest, the reclaimed page may be a DMA buffer. When a DMA request targeting the reclaimed DMA buffer arrives, the missing IOPT entry will cause DMA failure. To avoid potential DMA failures, the hypervisor needs to disable memory overcommitment, and statically pin the entire memory of a VM, *i.e.,* keep a fixed memory mapping for the VM during its life cycle.

2.2 Related Work

We discuss existing memory reclamation methods in the aspect of DMA-safety. We also characterize them in other important dimensions like compatibility and performance. We summarize them in Table 1.

2.2.1 I/O Page Fault

IOPF [25, 26, 34] is a hardware feature. When DMA failure occurs, IOPF allows a device to generate a page fault exception to the CPU, and replay the DMA request after the OS repairs the memory mapping. So by using IOPF, memory overcommitment is guaranteed to be DMA-safe.

As IOPF is transparent to the guest, it has no requirements for the guest OSes. However, it faces the issue of poor hardware compatibility as it requires designated hardware devices. Although the PCIe specification [34] has been extended to support IOPF since 2009, with the extensions of ATS (Address Translation Service) and PRI (Page Request Interface), the practice of such standard moves slowly. Currently, few off-the-shelf I/O device supports IOPF. Although manufacturers like AMD and Arm have introduced ATS and PRI to their SoC design [2, 8], the absence of PCIe devices that supports IOPF makes the IOPF call chain incomplete.

Even though the hardware compatibility issue can be fixed over time, the performance issue is unavoidable. Memory overcommitment depending on IOPF needs to backup/restore the memory content when doing memory reclamation/reallocation using memory swapping. Memory swapping introduces I/O overhead since the contents of the reclaimed memory are usually stored in storage media which is much slower than the main memory. Another method of memory swapping is memory compression. It introduces CPU overhead since the compress/decompress process is a heavy computing task. The overhead of memory swapping not only slows down the hypervisor's reaction to memory reclamation requests, but also causes performance oscillation to the VM and degrades the VM's SLO, especially when a VM faces a burst memory pressure and triggers a large number of page faults in a short time.

Meanwhile, the hypervisor's memory reclamation mechanism may also conflict with the memory reclamation inside the guest – The same memory page may be reclaimed twice, once by the guest and once by the hypervisor. In such a case, the guest's access to the memory page will trigger page faults twice, once trapped to the guest kernel and once to the hypervisor. This is the so-called *double paging* anomaly [14, 18, 33, 44] which introduces extra overhead to memory reclamation.

2.2.2 Monitoring DMA Buffers

Another way to avoid DMA failure is to monitor DMA buffer allocations in the guests. It implies two parts. First, when the hypervisor reclaims a guest page, the monitor tells whether it is a DMA buffer. Second, when the guest allocates a DMA buffer whose underlying page is reclaimed, the monitor notifies the hypervisor to fix the memory mapping in time. It is a software solution and does not rely on designated hardware.

vIOMMU [4] is one such solution. It exposes an emulated IOMMU (Input-output Memory Management Unit) to the guest and enables the hypervisor to intercept, monitor, and act upon DMA remapping operations. coIOMMU [40] is another one. It decouples the memory protection and pinning functionality in vIOMMU, which significantly improves the performance. But coIOMMU needs extensive changes to the guest OS, which results in the guest OS compatibility issue. At the same time, although monitoring guest DMA buffers guarantees DMA-safety, it still faces similar performance issues as the IOPF solution because of the need for memory swapping when doing memory reclamation/reallocation.

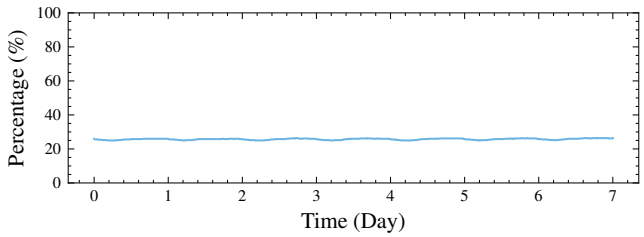


Figure 2: The free memory proportion of 10,000 VMs randomly sampled from the Alibaba cloud in one week period. The X-axis is the time, the Y-axis is the sum of the VMs' free memory size divided by the sum of the VMs' occupied memory size.

2.2.3 Free Page Reclamation

Free page reclamation targets on guest free pages. The quantity of guest free pages is considerable. Figure 2 shows the free memory proportion statistics of 10,000 VMs randomly sampled from the Alibaba cloud in one week period. As it shows, about 25.3% of the VMs' memory is free and this proportion is stable over time. At the same time, reclaiming free pages helps to eliminate the memory swapping overhead, since the contents of free pages are meaningless.

Ballooning [44] is a classical PV method that targets guest free page reclamation. It contains a frontend driver running in the guest, and a backend driver running in the hypervisor. The frontend driver can "inflate" to occupy the guest free pages and report them to the hypervisor, then the backend driver can reclaim those pages. The reclaimed GPA area is not allocatable in the guest until the hypervisor reallocates new pages for it. This can prevent reclaimed GPA areas to be used as DMA buffers, and thus avoids DMA failures.

However, the communication overhead between the frontend and backend drivers (Section 5.1.2) brings performance issues. Similar to the overhead introduced by memory swapping (Section 2.2.1), the communication overhead may also slow down the hypervisor's reaction to memory reclamation request, and cause VM performance oscillation [7, 9, 37].

Except for Ballooning, there are also other methods targeting free page reclamation. Free Page Reporting [45] allows the guest to report its free pages to the hypervisor periodically. And Hyperupcall [7], allows the hypervisor to inspect guest free pages without guest intervention. However, guaranteeing DMA-safety using these methods is a challenging task. As the memory reclamation action in these methods is transparent to the guest, the guest may allocate a free page, whose underlying physical page is reclaimed, as a DMA buffer. But the hypervisor will not be notified by this action, thus it does not have a chance to repair the memory mapping. So following DMA requests addressing this page will fail and cause DMA failures. Figure 3 illustrates the scenario. On the other hand, Free Page Reporting faces the problem of timeliness. Its frontend driver initiates the reporting procedure with a fixed delay (2 seconds in Linux's implementation) after the guest releases a high-order page. So the hypervisor's memory reclamation request may fail even if there are unreclaimed free pages in the guest.

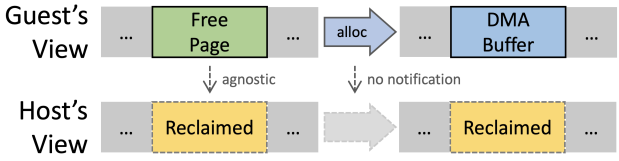


Figure 3: In guest free page reclamation, the hypervisor is agnostic to the DMA buffer allocation in the guest, and following DMA requests will cause DMA failures.

3 Approach

3.1 Design Goals

Based on our analysis in Section 2, we set our design goals as follows:

- **DMA-safety:** I/O passthrough is playing an irreplaceable part in today's virtualization systems since it significantly improves the guest I/O performance. The design should avoid DMA failures to make it safe for I/O passthrough enabled VMs.
- **Hardware Compatibility:** The design should not rely on IOPF as IOPF needs designated hardware. This ensures the solution is available for a large amount of existing hardware.
- **Guest Compatibility:** The design should be compatible with a wide range of commodity guest OSes. This makes sure the solution can be easily deployed to existing software systems.
- **Low Overhead:** The overhead of the solution should be as low as possible. This not only decreases the hypervisor's reaction time for memory requests and enables the hypervisor to take more aggressive memory reclamation decisions, but also limits the performance impact on the VMs and guarantees high VM SLO (Service-level Objective).

According to the summary in Table 1, memory reclamation methods that are unaware of the guest page allocation status (IOPF, vIOMMU, coIOMMU) require memory backup/restore for every reclaimed page, which introduces the overhead of memory swapping. However, obtaining knowledge of guest free pages usually needs communications to the frontend driver in the guest (Ballooning, Free Page Reporting), which introduces communication overhead. Hyperupcall

achieves low overhead, but it faces two challenges. First, as mentioned in Section 2.2.3, Hyperupcall is not DMA-safe. Although discarding guest communication makes the execution of Hyperupcall fast, this also means the guest is agnostic to the reclamation status of its free pages. The guest may allocate a reclaimed free page as a DMA buffer and the hypervisor is not notified by such behavior. It may not have a chance to fix the memory mapping before the DMA request arrives. Second, Hyperupcall relies on the complex toolchain of eBPF and needs modifications to the underlying LLVM compiler. This increases the deployment complexity of Hyperupcall, especially when adapting the toolchain to a variety of guest OSes in the public cloud.

Solving these challenges brings us to the solution of V-Probe. V-Probe is inspired by Hyperupcall. Like Hyperupcall, V-Probe allows the hypervisor to actively reclaim guest free pages without guest intervening. But V-Probe improves the programming model of Hyperupcall to overcome its complexity. More importantly, V-Probe proposes a novel fine-grained page meta-data management technique to guarantee DMA-safety.

3.2 V-Probe Overview

Figure 4 illustrates how V-Probe works. It contains three parts: (1) V-Probe injector, (2) V-Probe data manager, and (3) memory reclamation routine. Next, we will explain each part in detail.

3.2.1 V-Probe Injector

The V-Probe injector is a guest module. It only runs one time just after the guest finishes starting up. During its execution, it accomplishes two tasks: (1) page meta-data layout registration and (2) helper function registration.

In operating systems, like Linux, each physical page has a corresponding piece of page meta-data to record its usage state, *e.g.,* the reference count or the allocation status. And such meta-data is usually organized statically and continuously in ranges of the physical memory. In page meta-data layout registration, the guest first detects the GPA ranges where the page meta-data resides. Then it passes this information to the hypervisor through the registration API that V-Probe provides. As the hypervisor is aware of how GPA is translated to HPA, it can access the guest page meta-data precisely after obtaining the knowledge of guest page meta-data GPA ranges.

But only having access to the guest page meta-data is not enough, the hypervisor needs to understand the meanings of the bytes in it. So we need the helper functions which can parse the page meta-data. In helper function registration, the guest compiles the source codes of helper functions to hardware native binaries and injects them to the hypervisor through the registration API of V-Probe. As the formats of page meta-data can be different among guest OSes, the helper functions are guest-specific and need to be dynamically compiled in the guest.

3.2.2 V-Probe Data Manager

The V-Probe data manager runs in the hypervisor. It provides registration APIs for the V-Probe injector. The registered helper functions and page meta-data layout information are stored here. They help to recognize the status of each page in the guest when performing memory reclamation.

Since the hypervisor will directly call the helper functions via the injected binary code, we need a verifier to guarantee their integrity, *i.e.,* to prevent harmful codes injected from malicious guests. As the use case of the helper functions in V-Probe is limited to parsing guest page meta-data, the logic of the helper functions should be very simple. So the verifier uses strict rule-based restrictions to verify the injected binaries, which protects the hypervisor from security attacks without rejecting honest codes.

The V-Probe data manager also stores the information of the reclaimed memory set. Each record in the set represents a reclaimed guest memory area. Its content contains two dimensions: the reclaimed GFN (Guest Frame Number) range, and the GPA range for the corresponding page meta-data. This information is useful for page refaulting, which will be explained in the next subsection.

3.2.3 Memory Reclamation Routine

The memory reclamation routine is the core logic to reclaim free pages from the guest. Its execution is triggered by system events, like periodic timers, or memory pressure. After the preparation of the previous steps, the hypervisor can scan and parse guest page meta-data to inspect the status of each page.

V-Probe solves the DMA-safety problem via fine-grained page meta-data management. As shown in Figure 5, when reclaiming guest free pages, V-Probe not only invalidates the free pages' mapping in the MMU page table and the I/O page table but also modifies the page meta-data mapping in the MMU page table to read-only. The reclaimed memory set in V-Probe data manager will record the reclaimed GFN range, along with the GPA range of the corresponding page meta-data.

When the guest memory management system is going to allocate a page inside the reclaimed GFN ranges, it will first write to the corresponding page meta-data, *e.g.,* change the free flag. This will trigger a page fault on the CPU side as we have changed the memory mapping of the page meta-data to read-only after reclamation. The page fault notifies the hypervisor. Then the hypervisor looks for the corresponding record in the reclaimed memory set and recovers the memory mappings for both the pages and the page meta-data. This mechanism gives the hypervisor the chance to recover the

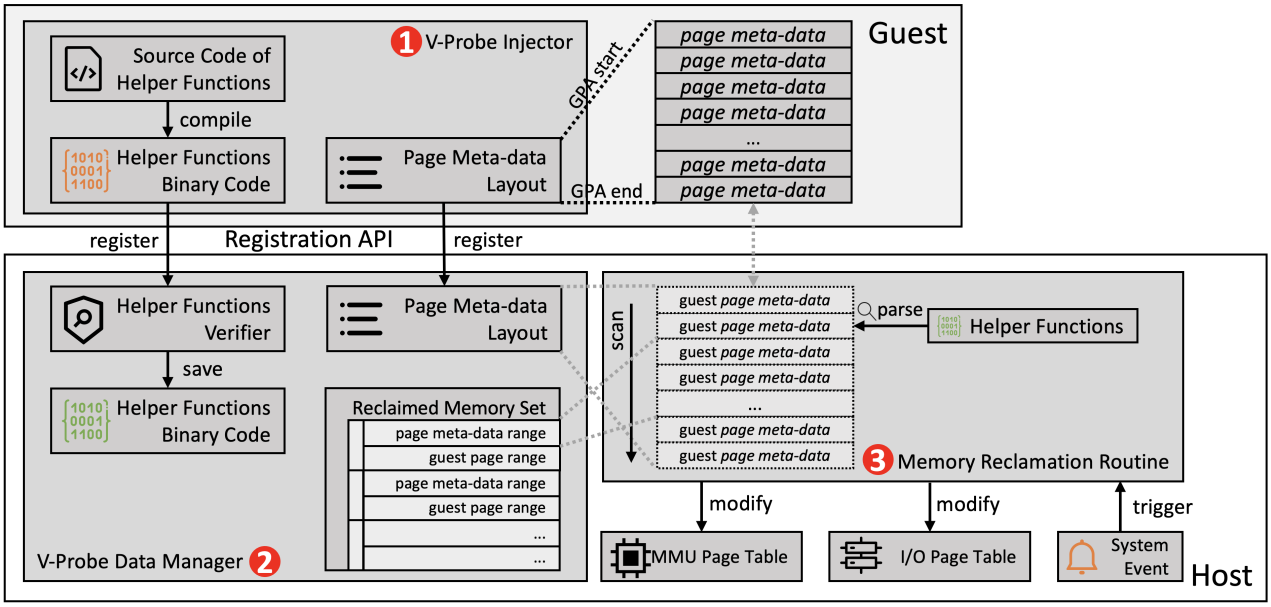
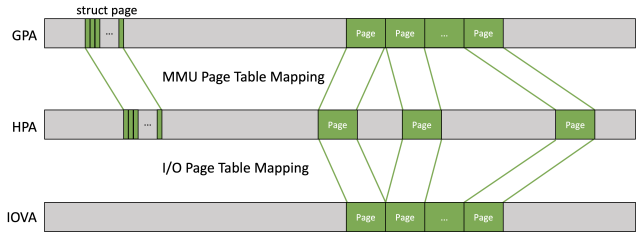
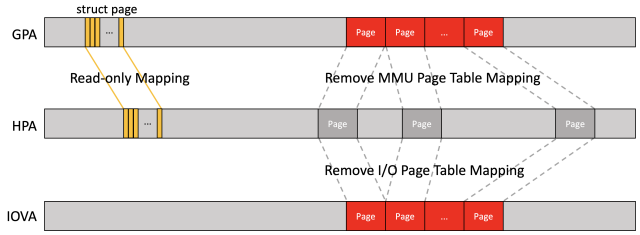


Figure 4: The design overview of V-Probe. It contains three parts: V-Probe injector, V-Probe data manager, and the memory reclamation routine. The V-Probe injector registers helper functions and page page-meta layout information to the V-Probe data manager. The code and data are used by the memory reclamation routine to inspect guest page status and reclaim free pages. The details of the design are explained in Section 3.2.



(a) Memory mapping before free page reclamation.



(b) Memory mapping after free page reclamation.

Figure 5: Memory mapping status before and after free page reclamation. During free page reclamation, V-Probe not only invalidates the page table entries for the reclaimed pages but also modifies the mapping of the page meta-data to read-only. This gives the hypervisor the chance to recover the memory mapping before the guest uses the page.

memory mapping before the guest uses the page, and thus it prevents DMA failures.

4 Implementation

In this section, we introduce the implementation details of V-Probe, which includes four parts: meta-data registration, helper function registration, memory reclamation routine, and page fault handler. We also explain the synchronization problem we face in V-Probe and how we solve this issue.

Our implementation of V-Probe is based on Linux, whose source code can be easily touched and modified. The hardware architecture is based on x86_64. The CPU we use is Intel® Xeon® Platinum 8163 CPU.

4.1 Meta-data Registration

The meta-data of pages in Linux is usually defined as type *struct page*. There are three kinds of *struct page* organization models in Linux, decided by one of the three compile configurations: `CONFIG_FLATMEM`, `CONFIG_DISCONTIGMEM` and `CONFIG_SPARSEMEM_VMEMMAP`. But no matter what the configuration is, the *struct page* data is organized linearly in ranges of continuous memory in the guest physical address space. V-Probe injector detects the *struct page* layout, indicating each *struct page* memory range with two dimensions: (1) the start and end GPA of the *struct page* memory range, and (2) the start and end GFN that are managed by the range

of *struct page*. V-Probe injector registers the information to the hypervisor, then the hypervisor can locate the *struct page* of each guest GFN.

4.2 Helper Functions Registration

```
noinline bool
page_free(struct page *page) {
        return PageBuddy(page);
}
```

```
8b 47 30          mov    0x30(%rdi),%eax
25 80 00 00 f0    and    $0xf0000080,%eax
3d 00 00 00 f0    cmp    $0xf0000000,%eax
0f 94 c0          sete   %al
c3                retq
```

(a) The source code and compiled binary of the helper function `page_free`. This function is used to parse whether a guest page is free. The function it calls (`PageBuddy`) is a Linux built-in function. The size of the compiled binary is only 17 bytes.

```
noinline unsigned long
page_order(struct page *page) {
    return page_private(page);
}
```

```
48 8b 47 28       mov 0x28(%rdi),%rax
c3                retq
```

(b) The source code and compiled binary of the helper function `page_order`. This function is used to parse how many free pages are adjacent to this page. The function it calls (`page_private`) is a Linux built-in function. The size of the compiled binary is only 5 bytes.

Figure 6: The source codes and their compiled binary of the two helper functions in V-Probe.

V-Probe relies on the helper functions to parse guest *struct page* and obtain the status of the pages. They are injected from the guest to the hypervisor after the guest finishes starting up. Since the formats of *struct page* vary from guests OSes, the helper functions need to be dynamically compiled in the guest.

Linux uses the buddy system [35] to manage free pages. It organizes free pages as blocks, and the size of one block is the order of 2. A buddy system block is identified by the leading *struct page* in the block, and it also records the order of the block. By parsing the *struct page*, *i.e.,* testing a specific bit or reading a specific field in it, we can decide whether a *struct page* is the leading one in a block, and get the order of the free page count in this block. In Linux, two kernel built-in functions, `PageBuddy` and `page_private`, help to do the parsing.

The two helper functions used in V-Probe are shown in Figure 6b: (1) `page_free`, which wraps the function `PageBuddy`, and (2) `page_order`, which wraps the function `page_private`. These two helper functions are short and simple, and the sizes of their binary code are also small: 17 bytes for `page_free` and 5 bytes for `page_order`. The definition of the helper functions requires the `noinline` decorator, to ensure they are not compiled to inline functions and can be called by the hypervisor in a function manner.

To avoid harmful codes injected from malicious guests, we need a verifier to guarantee the integrity of the helper function binary. We apply SFI to implement the verifier. As the logic of the helper functions used in V-Probe are very simple, simplified yet strict checking rules in SFI are enough to protect the hypervisor from security attacks without rejecting honest codes. These rules are as follows:

- Register `rdi` is read-only, register `rax` is read-write. Other registers are not allowed to be used.
- The function can only read a limited range of memory, which is usually the size of the guest *struct page*, pointed by the parameter stored in the `rdi` register.
- The function can not write to any main memory.
- No branch instruction or privileged instruction is allowed.
- The last instruction must be `retq`.
- The length of the binary is less than 64 bytes.

The above rules protect the hypervisor from being compromised by the injected code for two reasons. First, they make sure the helper functions obey the convention of a function call, and no branch nor privileged instructions are allowed. This implies the instructions inside the call will be executed sequentially, and the execution will finally return to the caller. Second, during the function call's execution, the accesses to registers and memory are strictly limited to avoid any host memory corruption. But note that these rules are highly related to the Intel X86_64 architecture. They need to be redesigned on other CPU platforms.

4.3 Memory Reclamation Routine

The memory reclamation routine is triggered by host events (like the memory pressure), and its execution is controlled by the hypervisor. Algorithm 1 shows the pseudocode of the procedure. It has two input parameters, *GUEST*, and *MIN_ORD*, which respectively indicate the target guest we try to reclaim memory from and the minimum order we want. The procedure iterates through each GFN in the guest (Line 2) and uses the two helper functions registered before, *GUEST.PAGE_FREE* and *GUEST.PAGE_ORD*, to check whether the guest page is free and what is the order in the buddy system (Lines 6-7). If it meets our requirements (Line 8), then we try to reclaim the corresponding pages (Lines

Algorithm 1 The procedure of memory reclamation

Input: $GUEST$, MIN_ORD
Output: $SUCCESS/FAIL$

```
1:  procedure MemoryReclamation
2:      for each GFN in GUEST do
3:          if GFN is reclaimed then
4:              continue
5:          SP ← struct page of GFN in GUEST
6:          FREE ← GUEST.PAGE_FREE(SP)
7:          ORD ← GUEST.PAGE_ORD(SP)
8:          if FREE and ORD ≥ MIN_ORD then
9:              Lock MUTEX
10:             Make the struct page GPA range read-only
11:             SP' ← struct page of GFN in GUEST
12:             FREE' ← GUEST.PAGE_FREE(SP')
13:             ORD' ← GUEST.PAGE_ORD(SP')
14:             if FREE' and ORD' ≥ MIN_ORD then
15:                 GFN_st ← GFN
16:                 GFN_en ← GFN + (1 << ORD')
17:                 RANGE ← [GFN_st, GFN_en)
18:                 Unmap EPT in RANGE
19:                 Unmap IOMMU in RANGE
20:                 Add RANGE to the reclaimed set
21:                 Release reclaimed pages to hypervisor
22:                 Unlock MUTEX
23:                 return SUCCESS
24:             Make the struct page GPA range read-write
25:             Unlock MUTEX
26:     return FAIL
```

Algorithm 2 The procedure of page fault handler

Input: $GUEST$, GPA

```
1:  procedure PageFaultHandler
2:      Lock MUTEX
3:      RANGE ← GetReclaimedRange(GUEST, GPA)
4:      PAGES ← pages reallocated for RANGE
5:      Map RANGE to PAGES in EPT
6:      Map RANGE to PAGES in IOMMU
7:      Remove RANGE from the reclaimed set
8:      Make the struct page GPA range read-write
9:      Unlock MUTEX
10:     return
```

Input: $GUEST$, GPA
Output: $RANGE$

```
11: procedure GetReclaimedRange
12:     for each RANGE in the reclaimed set of GUEST do
13:         SP_st ← the start struct page GPA in RANGE
14:         SP_en ← the end struct page GPA in RANGE
15:         if SP_st ≤ GPA ≤ SP_en then
16:             return RANGE
```

9-25). The core reclamation logic makes the *struct page* GPA range read-only (Line 10), invalidates the memory mapping in EPT (Extended Page Table) and IOMMU page table for the reclaimed GFN range (Lines 15-19), adds this range to the reclaimed set (Line 20), and finally, releases the reclaimed pages to the hypervisor (Line 21). Algorithm 1 uses the mutex (Line 9) and the page status double check logic (Lines 11-14) to avoid synchronization problems, which will be further explained in Section 4.5.

4.4 Page Fault Handler

Since we have modified the EPT mapping of the *struct page* GPA range to **read-only** for the reclaimed GFNs, when the guest tries to allocate these pages, it will **write** to the corresponding *struct page* to modify the bytes of page status flags. Thus a page fault is triggered. The pseudocode of the page fault handler is shown in Algorithm 2. It has two input parameters: $GUEST$ and GPA, which respectively indicate the guest and the accessed GPA that triggers the page fault. First, it calls *GetReclaimedRange* to find the reclaimed memory range responsible for GPA (Line 3). Then it allocates pages (Line 4), remaps the EPT and IOMMU page table for this range (Lines 5-6), removes the range from the guest's reclaimed memory set (Line 7), and resumes the EPT mapping of the *struct page* GPA range to **read-write** (Line 8).

Function *GetReclaimedRange* is simplified as a loop for ease of demonstration. We use the rbtree (Red-black Tree), which is an implementation of the binary search tree in Linux, to optimize the performance of memory range inserting, deleting, and searching.

4.5 Synchronization Problem

One challenge for the reclamation process is the synchronization problem: the guest may allocate the pages while we are reclaiming them, and the status of the guest *struct page* may change from "free" to "allocated" within the memory reclamation routine, which may cause inconsistency in the system.

Figure 7 show two cases to illustrate how the mutex and double-checking in Algorithm 1 (Lines 9-14) and Algorithm 2 (Line 2) avoids the racing conditions. In reclaim ①, if the guest allocates the free pages after the Hypervisor makes the corresponding *struct page* GPA range read-only, it will trigger page fault and the mutex will prevent the page fault handler to fix the memory mapping before the hypervisor finishes reclamation. This guarantees the atomicity of the memory reclamation process and the page fault handling. In reclaim ②, if the guest allocates the free page before the Hypervisor

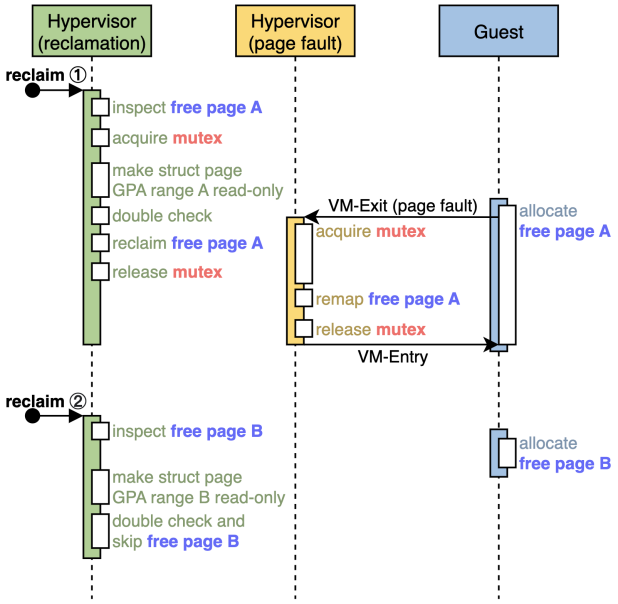


Figure 7: Two cases to illustrate how V-Probe solves the synchronization problem.

makes the corresponding *struct page* GPA range read-only, the double-checking will detect such behavior and skip the page, so as to avoid the inconsistency of the page status knowledge.

5 Evaluation

In this section, we conduct experiments to evaluate V-Probe in different aspects. First, we evaluate V-Probe's overhead in memory reclamation and reallocation tasks. Then we evaluate its impact on workload performance using both the micro-benchmark and macro-benchmark.

The experiments are based on the hardware architecture of Intel® Xeon® Platinum 8163 CPU. The hypervisor is based on QEMU and KVM. And the memory allocator manages the underlying memory of guest VMs in 2MB granularity. The guests in our experiments run Linux OS, each equipped with 2 CPU cores and 4GB main memory.

We use Ballooning and the IOPF based method for comparison. For simplicity, we will call the IOPF based method as IOPF in the following explanation. For memory reclamation and reallocation tasks, we compare V-Probe with both Ballooning and IOPF. For micro-benchmark and macro-benchmark tests, we only use IOPF for comparison. We do not use Ballooning for these benchmarks because the frontend driver of Ballooning will hold the reclaimed free pages and make them unallocatable in the guest. So if the memory usage of the task does not exceed the remaining allocatable memory in the guest, the performance will be the same as the baseline with no memory reclamation. On the other hand, if the memory usage of the task exceeds the remaining allocatable

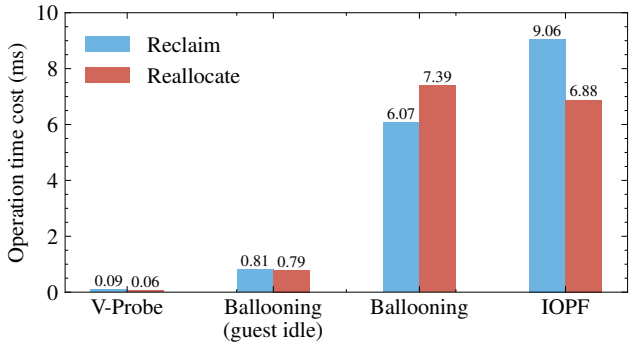


Figure 8: The time cost of V-Probe, Ballooning and IOPF to reclaim (reallocate) 2MB memory from (to) the guest. We run *sysbench* CPU workloads in the guest to emulate the guest's CPU stress.

memory in the guest, the guest will kill the task because of the OOM (Out of memory) error.

We reclaim 30% of the server's memory at the beginning of the each benchmark test. In practice, to avoid host memory insufficiency, the memory reclamation policies need to limit the quantity of memory reclaimed from each guest and the quantity of memory that can be reused by other VMs. 30% of memory reclamation is an extremely high value in practice. In this configuration, the benchmark results reveal the performance impact limit of V-Probe in a real-world environment.

Since IOPF is not commonly supported by off-the-shelf I/O devices, we implement IOPF by ourselves using FPGA-based SmartNIC. The storage media we use for memory swapping is a hard disk drive, with 250MB/s I/O throughput. Notice that other methods relying on memory swapping (vIOMMU, coIOMMU) share similar results with IOPF.

5.1 Overhead

5.1.1 Data Registration

The overall data registration process takes less than 1 second. The main overhead comes from the helper function compilation and page meta-data layout detection within the guest. The overhead of data transferring is small as the size of the registered data is only 232 bytes in our implementation, which includes the binaries of the helper functions, the page meta-data layout information, and the extra API-related fields. Also, as the helper functions are simple and small, the verification cost is negligible. Data registration only runs one time after the guest finishes starting up, and its overhead will not impact the following memory reclamation procedure.

5.1.2 Memory Reclamation and Reallocation

We evaluate the overhead of V-Probe in the memory reclamation (reallocation) task by measuring the time it takes to

reclaim (reallocate) 2MB memory from (to) the guest. For memory reclamation, the cost refers to the time elapsed between the initiation of the memory reclamation request and the moment the hypervisor releases the reclaimed memory back to the host. For memory reallocation, the cost refers to the time elapsed between the initiation of the memory reallocation request and the point at which the hypervisor establishes the new memory mapping.

We use Ballooning and IOPF for comparison. During memory reclamation (reallocation), we run *sysbench* CPU workloads in the guest to emulate the guest's CPU stress. Specifically, we run the command `'sysbench cpu -threads=$(nproc) -time=0 run'` in the guest to make all of the CPUs busy. We also assess the performance of Ballooning when the guest OS is idle, in order to determine the upper limit of its performance. For each measurement, we run the operation 10 times and present the average value as the result.

We will take the memory reclamation task as an example to analyze the experiment results. The memory reallocation task shares a similar analysis. As Figure 8 shows, V-Probe takes only 0.09ms to reclaim 2MB memory on average, while Ballooning and IOPF take 6.07ms and 6.88ms respectively. We analyze the reasons why V-Probe is two orders of magnitude faster than the other two methods as follows:

- Ballooning relies on the communication with the frontend driver running in the guest when reclaiming memory from it. The communication introduces a large delay, especially when the guest is busy with CPU intensive tasks, which reduces the chance for the thread of the frontend driver to gain CPU time slice. The overhead of communication (5.79ms) occupies 95.4% of the total overhead (6.07ms) when using Ballooning to reclaim 2MB memory. Even if the CPU is not busy in the guest, the communication overhead (0.53ms) also occupies 65.4% of the total overhead (0.81ms).

- IOPF relies on memory swapping to avoid guest memory corruption. Memory swapping introduces a large overhead of disk I/O (8.95ms), which occupies 98.8% of the total overhead (9.06ms).

- When triggered by events, V-Probe can use the injected helper functions to inspect the guest free pages actively, without the time-consuming communication with the guest. At the same time, since the reclaimed pages are free memory in the guest and do not contain valuable data, V-Probe does not need memory swapping, which avoids the disk I/O overhead.

The memory overhead of V-Probe's reclamation process is also small. For each reclaimed memory range, we allocate a 64B data structure to maintain the necessary data, which includes the start/end GPA of the *struct page* and the physical page, and other data-structure-specific fields. The overhead is 64B/2MB = 0.031‰ for a 2MB area and 64B/4MB = 0.015‰ for a 4MB area.

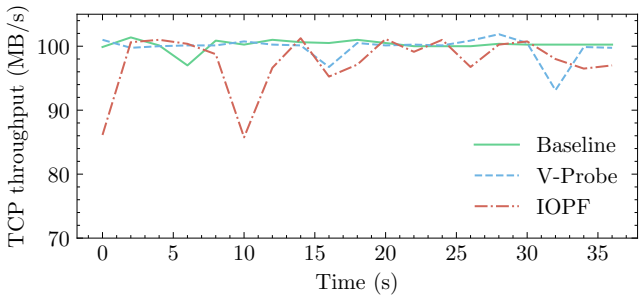


Figure 9: The TCP throughput of the baseline, V-Probe, and IOPF.

5.2 Micro-benchmark

The micro-benchmarks in our evaluation include TCP throughput and UDP latency, as these two metrics are critical for many important applications (*e.g.*, file servers, databases, and key-value stores). The evaluation requires a client-server experiment setting. We measure the performance impact of V-Probe on these metrics when a large amount of memory is reclaimed from the server side and high memory footprint is triggered by the network stream. We compare V-Probe with the baseline (*i.e.,* no memory is reclaimed from the guest) and IOPF.

Although there exist many network performance benchmark tools (*e.g.*, *netperf*), they only focus on the efficiency of the network stack and have a low memory footprint, which means they cannot cover the logic of the memory refault logic (*i.e.*, trigger page fault and reallocate memory). So we design and implement our own micro-benchmarks, which have a high memory footprint while measuring the TCP throughput and UDP latency.

5.2.1 TCP Throughput

To evaluate TCP throughput, we run a client and a server in two separate VMs, both equipped with 2 CPU cores and 4GB main memory. The server serves a 4GB file and the client will download the file when running the experiment. 30% of the server's memory is reclaimed at the beginning of the test. When the client downloads the file, the server will read the file from the disk to the in-memory page cache. As Linux drops the page cache in an on-demand manner, the server will fill the memory with the page cache until the memory is insufficient. This implements a high memory footprint of the experiment and will trigger page refault in the serve. We record the download speed as the TCP throughput.

Figure 9 shows the throughput changes over time in different settings. As the figure shows, the throughput fluctuation

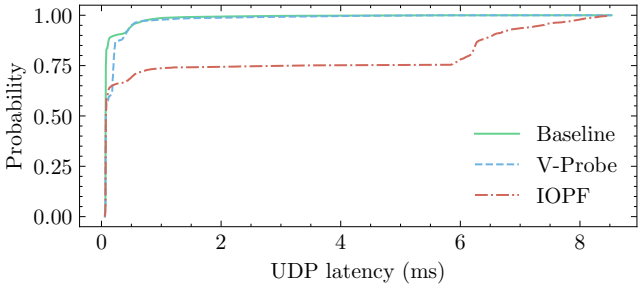


Figure 10: The CDF of the UDP latency when applying the baseline, V-Probe and IOPF.

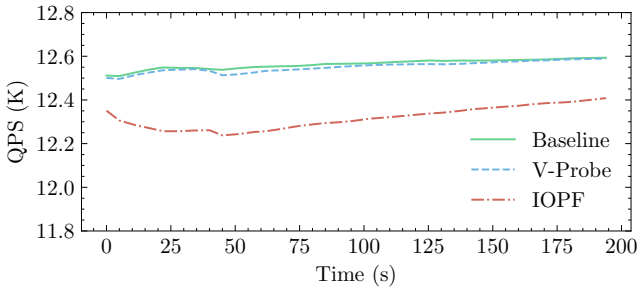


Figure 11: The Redis QPS when applying baseline, V-Probe, and IOPF.

of the baseline and V-Probe are smaller than that of IOPF. The minimum throughput of the baseline, V-Probe and IOPF are 97MB/s, 93.1MB/s, and 85.8MB/s respectively. And the average value is 100.2MB/s, 99.8MB/s, and 97.5MB/s respectively. These results show that V-Probe may degrade the TCP throughput slightly compared with the baseline, but it is much better than IOPF.

5.2.2 UDP Latency

To evaluate UDP latency, we run a client and a server in two separate VMs, both equipped with 2 CPU cores and 4GB main memory. 30% of the server's memory is reclaimed at the beginning of the test. The logic of the UDP server is quite simple: It listens on a port, accepts a UDP connection request, allocates a 2MB buffer and accesses it, then sends a response back to the client. This is to simulate a UDP application with a high memory footprint. Allocating and accessing buffers may cause page faults and page reallocations if the underlying memory is reclaimed by the hypervisor. The UDP client records the latency of each request as the results.

Figure 10 shows the CDF (Cumulative Distribution Function) of different settings. The 90th percentile latency of the baseline, V-Probe and IOPF is 0.23ms, 0.39ms and 6.55ms respectively, and the 99th percentile latency is 1.36ms, 2.43ms and 8.27ms respectively. The overhead of V-Probe is 69.6% and 79.4% in the 90th and 99th percentile latency, respectively, when compared to the baseline. However, it significantly outperforms IOPF, which introduces approximately 27X and 5X overhead in the 90th and 99th percentile latency.

5.3 Macro-benchmark

We use Redis, a widely used in-memory key-value database as the macro-benchmark to evaluate V-Probe. We run a client and a server in two separate VMs, both equipped with 2 CPU cores and 4GB main memory. At the beginning of the experiment, we remove all the data from the Redis server database and reclaim 30% memory from the server. The client runs `redis-benchmark`, the official Redis benchmark tool, to continuously send `SET` commands to the server. The `SET` command will randomly write key-value pairs to the in-memory database. There are 1,000,000 random keys, and the value size of each `SET` operation is 2KB. We record the QPS (queries per second) and the operation latency during the test. We only test the `SET` workload because V-Probe is a free-page-based method, and data insertion is necessary to trigger guest page allocation, and thus necessary to trigger V-Probe's memory reallocation. If we use the `GET` workload to evaluate V-Probe, the data in Redis will always stay in the main memory, with no page fault or memory reallocation operations will not utilize the functionality of V-Probe.

Figure 11 shows the QPS over time in the different settings. As it shows, the result of V-Probe (12.55K) is close to the baseline (12.56K). And the result of IOPF (12.31K) is about 2% lower than the other two. For the tail latency of each operation (not shown in the figure), the baseline and V-Probe exhibit a 99.99th percentile latency of 1ms, while the IOPF exhibits a noticeably higher 99.99th percentile latency of 7ms (the precision of the latency data is limited to the `redis-benchmark` tool). This indicates the Redis performance while applying V-Probe for memory reclamation is close to the case with no memory reclamation and better than applying IOPF.

6 Discussion

6.1 Compatibility

In this paper, we present a Linux-based implementation of V-Probe which is highly straightforward for two reasons. First, the V-Probe injector, including the helper functions, is lightweight (140 LOC (Lines of Code)), and it is dynamically compiled after the startup of the guest. This means that the V-Probe injector does not need to change for different releases of the Linux kernel. Second, V-Probe does not hack any API in the guest OS. This implies that V-Probe does not enforce any changes on the guest OS and retains full compatibility for guest OSes.

V-Probe relies on the continuous arrangement of the guest page meta-data. Accordingly, while in this paper we examine

a Linux-based implementation of V-Probe, the adoption of V-Probe is similar in Unix-like OSes that use a similar memory management mechanism to Linux. For example, FreeBSD uses ranges of *struct vm_page* (like *struct page* in Linux) to arrange the page meta-data and uses the buddy system to manage them.

V-Probe introduces about 1,300 LOC to the hypervisor, as the logic of V-Probe on the hypervisor side is much more complex. But this does not impact the practice of V-Probe as most of the cloud providers use customized hypervisors, and V-Probe can be easily integrated into these software systems.

V-Probe introduces high hardware compatibility as it only relies on the basic virtualization features of the CPU. These features are widely supported by popular CPU platforms such as Intel, AMD, Arm, and RISC-V. Furthermore, V-Probe does not rely on IOPF which is not commonly supported by off-the-shelf I/O devices.

6.2 Fine-grained Page Meta-data Management

Using fine-grained page meta-data management, V-Probe can effectively avoid DMA failures in I/O passthrough enables VMs. At the same time, the idea of fine-grained page meta-data management can also be combined with other free page reclamation methods to avoid DMA failures. For example, in Free Page Reporting, except for the free pages, the guest can also report the page meta-data to the hypervisor. So by managing the access mode of the guest page meta-data's GPA range, Free Page Reporting can avoid DMA failures in the same way V-Probe does. But unlike V-Probe, which uses helper functions to inspect guest free pages without notifying the guest, these methods rely on the frontend drivers to obtain the guest page status, which means the guest communication overhead is unavoidable.

6.3 Threats to Validity

There are two threats to the validity of our work. First, this paper defines and examines an example implementation of the V-Probe design that is specific to Intel X86_64 architecture and Linux-based OSes. Therefore, architecture-based and OS-based assumptions of this example implantation would need to be adjusted when moving to other architectures and OS types. However, since V-Probe relies on the continuous arrangement of the guest page meta-data, V-Probe may not apply to the OSes that use different ways to arrange the page meta-data.

Second, when reclaiming free pages, V-Probe needs to modify the memory mapping for the page meta-data GPA range. But since memory mapping is by the granularity of pages, the meta-data should reside on the same physical page. So V-Probe can only reclaim continuous free pages in batches. As a result, guest free page fragmentation may weaken the memory reclamation effect of V-Probe. But free page fragmentation in the guests is usually caused by high memory pressure, which suggests the system not to reclaim memory from them. Also, this problem can be mitigated by the memory compaction functionality in the guest OS.

7 Conclusion

In this paper, we conduct a systematic survey of previous VM memory reclamation methods and analyze their relationship with the widely deployed I/O passthrough technique. Based on the analysis, We propose V-Probe, a non-intrusion and efficient approach to achieve memory overcommitment for I/O passthrough enabled VMs. Using fine-grained page meta-data management, V-Probe enables the hypervisor to actively inspect and reclaim the guest free pages while guaranteeing DMA-safety. We implement V-Probe and evaluate its efficiency in different aspects. Experiment results show that the overhead of V-Probe is on the micro-second scale and it has low performance impact on the guest workload. It also has high compatibility with different hardware platforms and a wide range of Linux kernel releases, which simplifies the deployment.

References

[1] Abdullah Aljumah and Mohammed Altaf Ahmed. Design of high speed data transfer direct memory access controller for system on chip based embedded products. *Journal of Applied Sciences*, 15(3):576–581, 2015.

[2] AMD. Amd i/o virtualization technology (iommu) specification, 2021. https://developer.amd.com/wp-content/resources/48882_IOMMU_3.05_PUB.pdf.

[3] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. I/o paravirtualization at the device file boundary. *ACM SIGARCH Computer Architecture News*, 42(1):319–332, 2014.

[4] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, Assaf Schuster, et al. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 73–86, 2011.

[5] Nadav Amit, Abel Gordon, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Bare-metal performance for virtual machines with exitless interrupts. *Communications of the ACM*, 59(1):108–116, 2015.

[6] Nadav Amit, Dan Tsafrir, and Assaf Schuster. Vswapper: A memory swapper for virtualized environments. *Acm Sigplan Notices*, 49(4):349–366, 2014.

[7] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 97–112, 2018.

[8] Arm. Arm system memory management unit architecture specification, 2021. https://developer.arm.com/documentation/ihi0070/latest.

[9] Kapil Arya, Yury Baskakov, and Alex Garthwaite. Tesseract: reconciling guest i/o and hypervisor swapping in a vm. *Acm Sigplan Notices*, 49(7):15–28, 2014.

[10] S Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *2014 fourth international conference on advances in computing and communications*, pages 247–250. IEEE, 2014.

[11] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[12] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating System Design and Implementation (USENIX OSDI)*, volume 10, pages 423–436, 2010.

[13] 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*, pages 45–58, 2009.

[14] Khien-mien Chew and Avi Silberschatz. On the avoidance of the double paging anomaly in virtual memory systems. 1992.

[15] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[16] 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)*, pages 51–66, 2018.

[17] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, 2003.

[18] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 154–169, 1999.

[19] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. Proactively breaking large pages to improve memory overcommitment performance in vmware esxi. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 39–51, 2015.

[20] 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.

[21] Tai-Yi Huang, JW-S Liu, and Jen-Yao Chung. Allowing cycle-stealing direct memory access i/o concurrent with hard-real-time programs. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 422–429. IEEE, 1996.

[22] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[23] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016.

[24] Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu. Decoupled direct memory access: Isolating cpu and io traffic by leveraging a dual-data-port dram. In *International Conference on Parallel Architecture and Compilation (PACT)*, pages 174–187. IEEE, 2015.

[25] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. *ACM SIGARCH Computer Architecture News*, 45(1):449–466, 2017.

[26] Ilya Lesokhin and Dan Tsafrir. *I/O Page Faults*. PhD thesis, Computer Science Department, Technion, 2015.

[27] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (USENIX OSDI)*, volume 4, pages 17–30, 2004.

[28] Jiaxin Li, Dongsheng Li, Yuming Ye, and Xicheng Lu. Efficient multi-tenant virtual machine allocation in cloud data centers. *Tsinghua Science and Technology*, 20(1):81–89, 2015.

[29] 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.

[30] Duo Liu, Kan Zhong, Xiao Zhu, Yang Li, Lingbo Long, and Zili Shao. Non-volatile memory based page swapping for building high-performance mobile devices. *IEEE Transactions on Computers*, 66(11):1918–1931, 2017.

[31] 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 Twenty-Third ACM Symposium on Operating Systems Principles*, pages 115–128, 2011.

[32] Antonio Marsico, Roberto Doriguzzi-Corin, and Domenico Siracusa. An effective swapping mechanism to overcome the memory limitation of sdn devices. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 247–254. IEEE, 2017.

[33] KAZUHIKO OHMACHI, THORU NISHIGAKI, and SHIGEO TAKASAKI. Analysis of pawp/vms: Paging algorithm to prevent double paging anomaly in virtual machine systems. *J. Inform. Processing*, 4(2):55–60, 1981.

[34] PCI-SIG. Address translation services revision 1.1, 2009. http://www.pcisig.com/specifications/iov/ats/.

[35] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[36] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[37] Assaf Schuster, Nadav Amit, and Dan Tsafrir. Memory swapper for virtualized environments, November 7 2017. US Patent 9,811,268.

[38] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary {CPU} architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[39] SK Shravan, J Lakshmi, and Neeraj Bisht. Towards improving data center utilisation by reducing fragmentation. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 941–945. IEEE, 2018.

[40] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. coiommu: A virtual iommu with cooperative dma buffer tracking for efficient memory management in direct i/o. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 479–492, 2020.

[41] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. *Acm Sigplan Notices*, 50(7):1–15, 2015.

[42] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[43] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[44] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating System Design and Implementation (USENIX OSDI)*, pages 181–194, 2002.

[45] Wang Wei. Provide support for free page reporting, 2020. https://lwn.net/Articles/808807/.

[46] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[47] Zhen Xiao, Weijia Song, and Qi Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1107–1117, 2012.

[48] 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 *IEEE Symposium on Security and Privacy (SP)*, pages 79–93. IEEE, 2009.

[49] Pengfei Zhang, Xi Li, Rui Chu, and Huaimin Wang. Hybridswap: A scalable and synthetic framework for guest swapping on virtualization platform. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 864–872. IEEE, 2015.

[50] Qi Zhang and Ling Liu. Shared memory optimization in virtualized cloud. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 261–268. IEEE, 2015.

LPNS: Scalable and Latency-Predictable Local Storage Virtualization for Unpredictable NVMe SSDs in Clouds

Bo Peng
Shanghai Jiao Tong University

Cheng Guo
Shanghai Jiao Tong University

Jianguo Yao
Shanghai Jiao Tong University

Haibing Guan
Shanghai Jiao Tong University

Abstract

Latency predictability of storage is one important QoS target of the public clouds. Although modern storage virtualization techniques are devoted to providing fast and scalable storage for clouds, these works usually concentrate exclusively on giving high IOPS throughput without eliminating the device-level interference between multi-tenant virtualized devices and providing latency predictability for cloud tenants when the cloud infrastructures virtualize millions of the current commercially-available but unpredictable NVMe SSDs.

To resolve this issue, we propose a novel local storage virtualization system called LPNS to provide latency-predictable QoS control for hybrid-deployed local cloud storage, including virtualized machines, containers, and bare-metal cloud services. The OS-level NVMe virtualization LPNS designs reliable self-feedback control, flexible I/O queue and command scheduling, scalable polling design, and involves a deterministic network calculus-based formalization method to give upper bounds to virtualized device latency. The evaluation demonstrates that LPNS can achieve up to 18.72× latency optimization of the mainstream NVMe virtualization with strong latency bounds. LPNS can also increase up to 1.45× additional throughput and a better latency bound than the state-of-the-art storage latency control systems.

1 Introduction

Storage virtualization [23, 63] is critical to optimize limited hardware utilization and simplify storage management by providing consistent and straightforward I/O management interfaces in cloud systems. Since NVMe devices deployed in cloud platforms are usually inadequately utilized in terms of throughput [34, 35, 52], most previous works concentrated exclusively on achieving high-throughput targets, including software-level virtualization such as SPDK [25] and MDev-NVMe [55], the hardware-assisted virtualization such as the direct pass-through [70] and Single Root I/O Virtualization (SR-IOV) [14], and hardware/software co-design researches such as LeapIO [41] and FVM [37]. However, when more latency-critical businesses, in addition to the throughput-intensive businesses, have been migrating to the public cloud [12] for performance benefit, a fundamental contradiction between predictable latency and efficiency of the device sharing occurs when cloud platforms integrate storage virtualization: On the one hand, cloud service providers (CSP) tend to oversubscribe infrastructures by sharing them among multiple tenants to achieve better Input/Output Operations Per Second (IOPS) performance and higher energy efficiency [28, 44, 45]. On the other hand, the latency-critical tenants expect exclusive performance to ensure the latency bound of their services without worrying about interference from other guest machines that share the same NVMe SSD, i.e., latency-predictable Quality of Service (QoS).

The mainstream storage solutions can usually ensure high total throughput [25, 37, 41, 55, 57, 70] or fair bandwidth sharing [21, 71] but lack support for latency performance isolation between multi-tenant virtualized storage. For example, we quote a contention scenario where two virtual machines (VM1, VM2) share one Optane P5800X SSD [24] by using SPDK, and we use Figure 1 to show how the interference between these VMs hurts the average latency performance. In VM1, an IOPS-lightweight but latency-sensitive workload runs with a service level of agreement (SLA) at 30μs latency, while VM2 generates a throughput-intensive workload every 10 seconds as a competitor. Unfortunately, VM1 suffers a severe performance thrashing of over 250% additional latency as the workload weight of VM2 fluctuates, missing the SLA during its running time. The reason for this phenomenon is that the general storage controller naively handles all hardware queues used by different processes in a round-robin fashion [66], so the hardware queues are saturated with the I/O commands of the workload $w2$, resulting in a long queue operation time and unpredictable latency of $w1$.

In order to solve the performance interference issue and provide latency-predictable QoS, we propose LPNS, an NVMe virtualization solution that provides latency-predictable virtualized storage in NVMe virtualization and sharing scenarios. LPNS replaces the original static I/O queue allocation be-

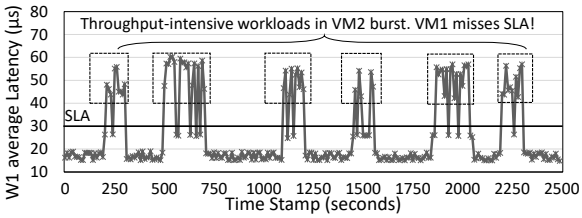


Figure 1: The workload $w1$ in the VM1 interfered by intensive workloads $w2$ in the VM2 sharing the same SSD. The $w1$'s average latency misses the 30μs SLA.

tween hardware and virtualized devices with dynamic queue scheduling. LPNS introduces a fine-grained I/O command scheduling to throttle the competing throughput-intensive workloads to constrain the latency of latency-sensitive workloads. Moreover, we have pioneered the introduction of deterministic network calculus [5, 38] into LPNS to provide mathematical modeling for the latency predictability of virtualized storage, giving a definite latency bound by abstracting the arrival/service curves for storage systems. The experiments show that LPNS can guarantee the latency-predictable QoS and achieve up to **11.57/18.72**× latency optimization over the SPDK/SR-IOV by eliminating the device-level interference. LPNS can also achieve better latency bounds (50μs) than the state-of-the-art mechanism K2 [48] (80μs) for real-world I/O trace on P5800X and increase up to **1.45**× additional total throughput over K2 (**1.61GB/s**, equivalent to **47.66%** of the maximum throughput of a P5800X SSD).

To sum up, we make the following contributions:

(1) We analyze the device-level latency interference of the commercially-available but unpredictable NVMe SSDs, and we argue the significance of overcoming this interference from the OS-level storage virtualization design aspects.

(2) We design LPNS, the first OS-level NVMe virtualization solution with latency-predictable QoS enhancement for unpredictable NVMe SSDs in clouds. LPNS designs a self-feedback mechanism that adaptively provides predictable latency according to the workload distributions of multi-tenant VMs. LPNS involves deterministic network calculus to verify the latency upper bound.

(3) We implement LPNS based on mediated pass-through to enhance the original Linux NVMe driver in providing latency-predictable QoS for hybrid-deployed local virtualized and cloud-native storage. All the CSPs can directly use the OS-level LPNS to provide latency-predictable NVMe storage virtualization and sharing without hardware modification and purchase costs.

(4) The evaluations prove the effectiveness of the latency-predictable QoS control of LPNS, compared with previous storage virtualization and other latency QoS control solutions.

The rest of the paper is organized as follows: Section 2 introduces the technical backgrounds of NVMe, cloud storage, and network calculus. Section 3 introduces the motivation for designing scalable and latency-predicable cloud storage virtualization. Section 4 describes the design and implementation of LPNS. Section 5 demonstrates the evaluation results of LPNS. Section 6 introduces the related works, and Section 7 concludes this paper.

2 Background and Motivation

2.1 NVMe Storage

The NVMe SSDs are now widely used in public clouds. The NVMe specification [51] is an efficient and scalable interface designed for high-performance SSDs. NVMe supports up to 65,535 I/O queues whose depth can be up to 65,535. Specifically, each queue pair contains a submission queue (SQ) and a completion queue (CQ). During each I/O execution, the host OS stores IO commands into the SQ and rings the doorbell, and the completion messages are placed into the corresponding CQ by the SSD controller. With the high-parallel SQ/CQ interaction between the host and the SSD controller, NVMe SSDs can obtain high throughput and micro-second-level latency advantages over the traditional interfaces [9], wherein both throughput and latency are extremely significant and mutually restrictive QoS targets of the cloud storage systems.

2.2 Local NVMe Storage for Cloud Services

For better performance and resource utilization, public cloud infrastructures usually adopt two types of solutions to manage their millions of NVMe SSDs. One is running cloud instances or workloads directly on the native servers and using the local storage; another is providing efficient data access to a remote storage pool or dedicated storage servers [36, 41, 46].

However, not all cloud services prefer remote storage solutions, for example, Elastic Compute Services (ECS). We infer there are three main reasons: (1) The remote storage performance is influenced not only by the storage system but also by network devices, which introduces an additional bottleneck of latency performance incurred by the network. (2) Remote storage uses expensive network devices, incurring additional purchase costs to cloud infrastructures and finally hurting the interests of the cloud tenants. (3) Cloud tenants may lease the bare-metal servers or services to customize their own distributed computing and storage clusters, which conflicts with the remote storage architectures.

In contrast, the local storage technique route can provide fast and cheap storage for the public clouds with the widely-used storage virtualization [7, 25, 37, 55, 57, 70]. For example, MDev-NVMe [55] proposes a novel I/O queue pass-through of NVMe hardware queue resources to achieve near-native performance for cloud instance storage. Moreover, local storage virtualization is more flexible in providing QoS guarantees for cloud services, especially the latency-sensitive services that suffer the performance unpredictability of network systems. Therefore, in this paper, we aim to provide latency-predictable virtualized storage services by following the local

storage virtualization technique route.

2.3 Deterministic Network Calculus

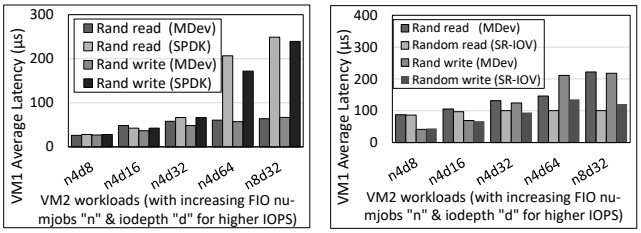
Deterministic Network Calculus (DNC) [38] is used to calculate theoretical worst-case performance guarantees for networks of queues and schedulers, which is commonly used for communication networks to provide predictable latency in typical deterministic queuing systems [13, 40, 59, 65, 69]. The theory's three basic concepts are suitable for providing predictable-latency performance in NVMe virtualization: the arrival curve, the service curve, and the virtual delay. The arrival curve expresses the upper bounds of the number of events that come from the sources over any time. The service curve refers to the guarantee of flows offered by the system and describes the service capability of the system. For a definite system, if the arrival curve and service curve are determined, the virtual delay referring to the delay that an event arriving at a particular time will suffer, can be deduced. The deterministic network calculus proves that the maximum horizontal distance between a workload's arrival curve and service curve is a tight worst-case bound on latency.

According to the NVMe specification and the I/O performance and behavior of the commercial-available NVMe SSDs, we can make a performance assumption that the processing capability of the modern NVMe SSD is stable and constant at most of their working time except during garbage collection and the SSDs serve the I/O command processing in First-In-First-Out (FIFO) policies. Moreover, for the multi-tenant virtual devices sharing the same NVMe SSD, we assume that throughput-intensive workloads can use the maximum queue depth and latency-sensitive processes use a queue depth of 1. The arrival curve refers to the actual IOPS of VM workloads, which determines the commands rate of multiple-tenant workloads that the SSD receives. The service curve guarantees the least command rate that the SSD can process during a busy period. The virtual delay is precisely the I/O latency of the latency-sensitive workload, which is the focus of our attention for latency-predictable QoS in storage virtualization.

3 Motivation

The clouds aim to provide **latency-predictable Storage QoS** for VMs so their virtual storage devices can have a latency bound for storage I/O operations. However, state-of-the-art storage virtualization [14, 25, 37, 41, 55] cannot solve the device-side latency interference issue and cannot provide latency-predictable QoS. The **device-side latency interference** refers to a phenomenon that the guarantee for the VM with latency-predictable QoS fails when multiple VMs without latency-predictable QoS run throughput-intensive workloads and compete for the same underlying NVMe SSD, incurring the latency deterioration, unbounded latency, and the miss of latency QoS (or SLA) just like the Figure 1 example.

However, even the state-of-the-art NVMe virtualization techniques (including SPDK, SR-IOV, and MDev-NVMe) neglect the importance of eliminating performance interference in multi-tenant storage-sharing scenarios. The more intensive the competitor's workload is, the more severe performance interference happens. We use MDev-NVMe and SPDK vhost-blk to share one Intel Optane P5800X SSD into two virtualized devices. We also use one SR-IOV-capable Samsung PM1735 to build two VFs and use MDev-NVMe as a comparison. In the test cases, VM1 runs a latency-sensitive workload (an FIO [26] random read or write benchmark with *iodepth=1, numjobs=1*), with a growing-intensive 4K random write workload in VM2 by increasing the *numjobs* and *iodepth* parameters of FIO. We separately show the latency performance of the latency-sensitive workload of VM1 in Figure 2, which shows that there is an up to 4.7× latency overhead in MDev-NVMe, 16.5 × overhead in SPDK, and up to 6.1× overhead in SR-IOV over the VM1 workload.



(a) Intel Optane P5800X (b) Samsung PM1735

Figure 2: Latency interference between two virtualized devices with the state-of-the-art NVMe virtualization. (If VM1 monopolizes one P5800X, the read/write latency of the VM1 workload is 11.9/13.1μs with MDev and 11.9/13.3μs with SPDK. If VM1 monopolizes one PM1735, the read/write latency is 71.9/17.6μs on MDev and 74.9/19.9μs with SR-IOV.)

We further analyze the latency distribution of different I/O phases of NVMe virtualization. We choose MDev-NVMe as a representative to virtualize a P5800X and summarize the results in Table 1. We find that the VM1's average latency on the NVMe controller grows from 62.5% to **93.0%** of the total latency with the increasing IOPS of the VM2 workload. This phenomenon proves that more severe I/O congestion happens when more commands from the competitor workloads simultaneously arrive at the SSD controller and savagely preempt the resources, causing a worse latency bound to the latency-sensitive workloads. Since the hardware/software co-designed virtualization [37, 41] usually attaches standard and unpredictable NVMe SSDs to an accelerator card, we can deduce that these solutions still meet this device-side latency interference and fail to reach latency-predictable QoS.

To overcome the device-level latency interference, we aim to design latency-predictable QoS control for NVMe virtualization. Previous works (summarized in Table 2) usually redesign the Flash Translation Layer (FTL) in the SSD con-

Table 1: VM1 latency distribution in different phases

I/O phase \ VM2 IOPS	50K	250K	500K
Guest OS submit commands	2.8%	1.9%	1.1%
Virtual SQ	1.9%	1.9%	0.8%
Physical SQ	1.4%	1.1%	0.4%
SSD controller	**62.5%**	**80.1%**	**93.0%**
Virtual CQ	23.0%	10.2%	2.9%
Virtual Interrupt handling	8.4%	4.8%	1.7%

troller [29,31,53,66] or introduce additional logic into the host software stack [21,48,71] to achieve reliable high-throughput or fair-bandwidth QoS control. FinNVMe [54] local NVMe virtualization designs fine-grained queue-level scheduling to achieve state-of-the-art throughput-oriented QoS control for virtualized devices. For predictable latency, Prioritymeister [74] automatically and proactively configures workload priorities and rate limits to provide tail Latency QoS for shared networked storage. K2 [48] uses work-constraining scheduling to trade reduced throughput for lower latency bound, which is the state-of-the-art latency QoS control among the previous solutions [15,22,29,30,33,36,50,61,64] for native storage. However, Prioritymeister and K2 lack the customized design for NVMe virtualization in multi-tenant cloud storage systems. Moreover, K2 sacrifices too much throughput (up to **2.27GB/s,** equivalent to **47.66%** of the maximum throughput of the P5800X SSD in Section 5 experiments) when reaching predictable latency.

Table 2: Storage resource sharing and scheduling systems.

Systems	Virtualization Optimized	QoS Control	Predictable Latency
VirtIO [57], SPDK [25]	PV	✗	✗
MDev-NVMe [55]	MPT	✗	✗
FinNVMe [54]	MPT	✓	✗
WA-BC [29], LeapIO [41], FVM [37]	SR-IOV	✓	✗
AutoSSD [31], FIOS [53], FLIN [66]	N/A	✓	✗
K2 [48]	N/A	✓	✓
MQFQ [21], D2FQ [71]	N/A	✓	✗
LPNS (Our work)	MPT	✓	✓

*PV: Para-Virtualization. MPT: Mediated Pass-through.

4 LPNS Design and Implementation

4.1 System Overview

Motivated by the analysis of device-side latency interference, we aim to provide predictable latency and overcome the interference problems from the aspect of the OS-level NVMe virtualization design. The QoS levels of different VMs should be determined at initialization and can only be changed when storage service tenants agree. Since the strict predictability generalizing the notion of isolation comes at the overall throughput expense, we should give an upper bound to the latency of NVMe virtualization under definite system settings with a slight total throughput loss of the SSD.

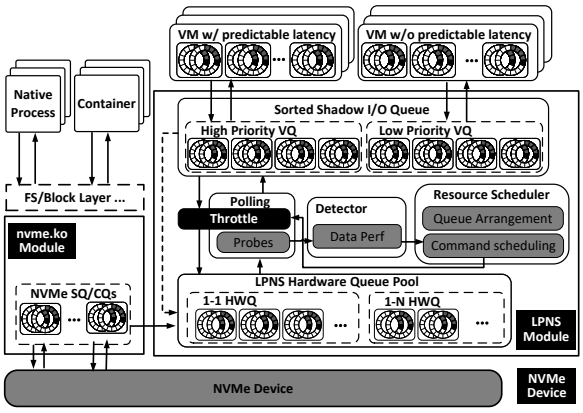


Figure 3: The system architecture of LPNS.

Scalable architecture. We briefly introduce the LPNS architecture in Figure 3. LPNS is designed based on mediated pass-through [27], which has been proven to be excellent in both performance and scalability [54,55,67]. LPNS is implemented as a kernel module to provide virtualized storage with full NVMe features to guest VMs, and it can coexist and cooperate with the original *nvme.ko* module (the NVMe driver) to support the **hybrid deployment of host processes, containers, VMs on each single NVMe SSD**. So it is cheaper, more flexible, and user-friendly for cloud vendors to use LPNS than SR-IOV-capable SSDs or the hardware/software co-designed solutions. Specifically, LPNS designs a performance detector, a queue scheduler, and a command scheduler for predictable latency enhancement, and it provides a flexible polling mechanism for better virtualization scalability.

Full virtualization. LPNS inherits the advantages of physical I/O queues pass-through and active I/O polling from the previous NVMe mediated pass-through solution [55]. LPNS supports full virtualization and does not modify the guest drivers. In the hypervisor (kernel module), the hardware I/O queues (HWQs) can be directly passed-through to VMs, so I/O commands from VMs can be stored in the HWQs through fast I/O paths. The hypervisor maintains a virtual IOMMU structure in shared memory for translating the GPA (Guest Physical Address) of different virtual devices into the IOVA (IO virtual address) of the underlying SSD. Cloud vendors can create partitions and bind each partition with a virtualized storage device with the hypervisor, and the hypervisor can easily do LBA translation between guest and host OS based on the partition information.

Self-feedback QoS control. LPNS has the ability to distinguish QoS targets of VMs to provide latency-predictable QoS. The hypervisor gives each virtual storage a tag when creating the VM to recognize if the workload from this VM should be provided with a predictable latency guarantee [1]. LPNS can periodically trigger resource scheduling between VMs based on runtime performance detection. We place a

[1]We use SVM to represent the VM with latency-predictable QoS guarantees, and use IVM to represent the VM without latency-predictable QoS.

performance probe in the polling thread to periodically collect the index, submission, and completion timestamps of all virtual I/O commands, along with real-time submission and completion command counts. With these statistics, LPNS can calculate the primary execution time of every command and the average latency within a time interval for the scheduler to enhance predictable latency and build an entire self-feedback QoS control system. Specifically, we choose 10 ms as a monitoring interval and 200 ms as a scheduling period in the current implementation.

Flexible and scalable polling. LPNS uses polling threads to process I/O commands and poll the virtual SQ (VSQ) tail, the HWQs, and the virtual CQ (VCQ) head of all VMs for better I/O performance. To reach a balance between performance optimization and CPU overhead, LPNS can arrange only one polling thread for all IVMs, and the thread can fully utilize the high throughput ability of the NVMe SSD; or arrange one dedicated polling thread for each SVM to reach a predictable latency (performance discussed in §4.5). All the polling threads are adaptively triggered on or off according to runtime workload detection of the VMs to reduce unnecessary CPU overhead.

Hardware queue pools (§4.2.) LPNS designs an I/O queue scheduling mechanism with a dynamic queue allocation to achieve a flexible mapping and multiplexing of hardware I/O queue resources. All HWQs are organized into a queue pool as two types of queues, wherein the 1-1 HWQs are exclusive queues for one single VM, and the 1-N HWQs are the shared queues for multiple VMs. The hypervisor implements an HWQ scheduler as a global controller for the HWQ resources. The queue scheduler can periodically schedule the 1-N HWQs between VMs for better virtualization scalability.

I/O command throttling (§4.3.) We implement a virtual I/O command throttling mechanism in the LPNS hypervisor to control the I/O path of each VM and eliminate the device-level latency interference at the OS level. The polling threads can perform the throttling between VMs from the global view. Specifically, the hypervisor provides an interface to adjust the threshold for command throttling, which can control the I/O rate received by the hardware at each scheduling period to reach predictable latency guarantees. The I/O command throttling follows the constraint of deterministic network calculus.

4.2 Scalable I/O Queue Handling

Since previous NVMe virtualization solutions usually use static I/O queue shadowing between virtualized devices and the hardware SSD, the total number of HWQs exposed by the SSD will limit the maximum number of VMs sharing the same underlying SSD. To increase the virtualization scalability, LPNS supports the flexible remapping between HWQs and virtual queues (VQ). Specifically, LPNS can allocate any number of the HWQs (but less than the maximum number of HWQs exposed by the SSD controller) from the *nvme.ko* kernel module into a **Hardware Queue Pool** for I/O queue scheduling, and the rest HWQs can be used by the native applications and containers.

We design an **I/O queue scheduler** to manage the Time Division Multiplexing (TDM) [16] of the HWQs in the **Hardware Queue Pool**. We abstract the HWQs of the pool into two types: 1-1 HWQs and 1-N HWQs. An 1-1 HWQ refers to an HWQ that can only be bound to one VQ. The 1-N HWQs refer to the I/O queues to maintain the necessary I/O capabilities for the other multiple VQs. The total number of 1-1 and 1-N HWQs should be configured when the host system initializes the LPNS module. Specifically, the configuration will not directly change the priority of these HWQs, so it can still work when using the NVMe Weighted-Round-Robin-with-urgent-priority (WRR) feature of the HWQs.

When multiple SVMs and IVMs share the same underlying SSD, LPNS only assign 1-1 HWQs to the SVMs for better latency performance, so the number of 1-1 HWQs should not be less than the number of total VQs owned by all the SVMs. The queue scheduler can schedule the idle 1-1 HWQs and all the 1-N HWQs among the other IVMs.

Since LPNS can monitor and collect real-time performance and workload data of VMs in each period, the I/O queue scheduler follows a hierarchical **workload-aware HWQ scheduling** policy wherein it takes the QoS target and runtime workloads of VMs as the algorithm inputs. The scheduling mainly consists of a hierarchical VQ weight calculating phase and an HWQ switching phase. During each scheduling period, the scheduler first respectively calculates the weight of VQs of all VMs. For any SVM VQ, the weight is set as 0 or the top weight, depending on if this VQ is empty or not. For the VQs of IVMs, their weights are equal to the number of their backlogged commands so that VQs with heavier workloads can get higher priority to be drained quickly. The HWQ switching phase works after the weight updating phase. It sorts the weights of all VQs in descending order, and those high-priority VQs will first use 1-1 HWQs, and the low-priority VQs will be bound to 1-N queues. After switching, the I/O queue scheduler will sleep until the next period.

When one 1-1 HWQ needs to switch to a new VQ, it may still have unfinished commands from the former VQ. Direct forwarding of these stranded commands in VSQs will cause an I/O error because the completion message cannot be handled correctly. So we design a seamless switching mechanism in the I/O queue scheduler. Specifically, LPNS extends the virtual NVMe command structure with an index of the VM to enable the HWQs to interact with different VQs from different VMs simultaneously in the seamless switching. When we want to unbind a 1-1 HWQ from a VQ, the VQ should be bound to another backup 1-N HWQ before the commands in the 1-1 HWQ are executed by the SSD. During the switching, both the 1-1 HWQ and the backup 1-N HWQ can write back the completion information into the original VCQs. Specifically, the VCQ should be locked to ensure data consistency

when two HWQs access the same VCQ simultaneously. The seamless switching operations run in the background, and each VQ can continuously send commands to an HWQ without the perception of queue scheduling operations until a new HWQ-VQ binding relationship is established. So it will not disturb the high-parallel fetch of I/O commands from VMs.

4.3 I/O Command Throttling

We design a fine-grained I/O command throttling in LPNS and involve a deterministic network calculus to provide a latency bound of the I/O command for multi-tenant shared virtualized storage. Network calculus has been proven effective in providing latency control for shared network storage systems in previous research, such as Prioritymeister [74]. In our LPNS, we use deterministic network calculus to help to solve the intensive resource and performance competition for different virtual devices sharing the same SSD so that a storage hypervisor can directly eliminate the device-level latency interference from the OS and virtualization layer by scheduling queue resources between different virtual devices and control the I/O command throttling inside a kernel module.

The deterministic network calculus gives a definite bound to the command latency by abstracting the arrival and service curves for storage systems. We can fix the I/O block size to the most widely-used 4K for simplification of the predictable latency deduction. The arrival curve expresses the upper bounds of the number of events from the sources over any time. It refers to the I/O command submission rate by all VMs in our system. The service curve refers to the guarantee of flows offered by LPNS wherein it describes the I/O capability of the NVMe devices. The service curve may be modified by internal functions (like garbage collection and block relocation) of SSDs, so we simplify the mathematical model of LPNS by concentrating on the normal working time without triggering internal functions. And we believe LPNS can cooperate with some future SSDs, such as AutoSSD [31], that try to control the tail latency inside the SSD controller so that LPNS can provide a more robust latency-predictable storage virtualization on the future NVMe SSDs. Once the arrival and service curve of an NVMe SSD is determined, we can deduce the virtual delay at a particular time t, which is the latency bound of SVMs in LPNS virtualization.

Arrival Curve. If the total command submission rate of an intensive VM exceeds θ times the slowest submission rate of a VM with predictable latency QoS guarantee, its I/O will be suspended. The polling threads in the hypervisor of LPNS can trigger the suspension based on the command count of each VM recorded by the performance detector. Using this I/O command throttling threshold is incredibly effective in guaranteeing the latency stability of the VMs with predictable latency guarantees because we can control the command submission rate precisely as we expect. Suppose there are j VMs with intensive workloads co-running with i VMs running latency-sensitive workloads whose IOPS is p and its command queue depth d , the real-time total command submission rate sent to the SSD hardware is:

$$v = p \cdot (j \cdot \theta + i), \tag{1}$$

where the I/O command submission rate v of LPNS is proportional to the IOPS of latency-sensitive workloads.

Similarly, we can get the number of commands b that the sources can send at one time:

$$b = d \cdot (j \cdot \theta + i). \tag{2}$$

Then we formulate the arrival curve of LPNS as:

$$\alpha(t) = v \cdot t + b = (p \cdot (j \cdot \theta + i)) \cdot t + d \cdot (j \cdot \theta + i). \tag{3}$$

Service Curve. The service curve is straightforward because the processing capability of the NVMe SSD is constant (according to the types of NVMe SSDs). We use R to represent the parallel speed of which the hardware processes random write commands per second (since most SSDs has lower random write performance than random read), and L_h to represent the minimum completion latency of an I/O command. So the service curve is:

$$\beta(t) = R \cdot t + L_h. \tag{4}$$

Latency Upper Bound. Call $\Delta(t) = \inf\{\tau \geq 0 : \alpha(t) \leq \beta(t+\tau)\}$. Let $h(\alpha,\beta)$ be the supremum of all values of $\Delta(t)$, then the virtual delay for all t satisfies: $L(t) <= h(\alpha,\beta)$. Given the arrival curve and service curve of LPNS above, we deduce the upper bound of latency $L_{\max}$ as (according to [5, 38]):

$$L_{\max} \leqslant b/R + L_h = d \cdot (j \cdot \theta + i)/R + L_h, \tag{5}$$

where we let $\Omega = j \cdot \theta + i$ to control the predictable latency performance of a latency-sensitive workload p and the total submission rate (which can finally decide the total throughput) on each type of the NVMe SSDs. Specifically, LPNS can adaptively change the θ parameter with the numbers of VMs (i and j) when the performance detector checks if each VM generates active I/O operations when the Ω is determined.

The choice of threshold Ω is essential to the effect of throttling in the real-world system. A smaller Ω value can provide better predictability, but it restricts the throughput of the VMs running throughput-intensive workloads without predictable latency requirements. So the most proper alternative of Ω should be figured out by the optimal upper limit of the hardware processing speed, which needs to be specified and updated by the system administrator because different NVMe SSD may have various R/W performances. Moreover, the requirements for the latency QoS level of each latency-sensitive workload can be stricter and looser in practice, so as the constraints for Ω. Therefore, LPNS can let VMs introduce a tenant-defined latency target in advance and help to tune the Ω parameter. And the decision-maker can increase

or decrease Ω by comparing the detected latency with the target during the running time for a better trade-off between throughput and predictable latency. In the evaluation section, we will verify the effectiveness of this latency bound through adequate experiments on different types of NVMe SSDs.

4.4 Latency-Predictable I/O Processing

We use Figure 4 to represent how LPNS enhances latency-predictable I/O processing of the SVMs. The left part of this figure shows the I/O path of a VM with the latency-predictable QoS (the SVM). The LPNS module maintains the shadow I/O queue data structure (directly corresponding to the virtual queue) for each virtual storage in the shared memory between the host kernel and QEMU, which can store I/O commands from guest SQs and generate completion messages into the guest CQs. When guest applications generate I/O operation on the virtualized storage (❶), the command will be stored in the shadow I/O queue. The polling thread will immediately poll the head of the queue (❷), and translate the DMA and LBA addresses in commands, store it into the 1-1 HWQ, and finally ring the hardware doorbell register. (❸) - (❺) represents the process that the SSD controller fetches command, generates DMA, and stores the completion messages into HWQs. The polling thread will continuously check if the doorbell register of the hardware CQs updates to accelerate the I/O operation instead of waiting for the SSD controller to inject interrupts (❻). And the polling thread will compose the completion message of the VMs and store the message into the shadow queue (❼), and inject a virtual interrupt into the VM. Finally, the guest VM driver can complete the I/O operation (❽).

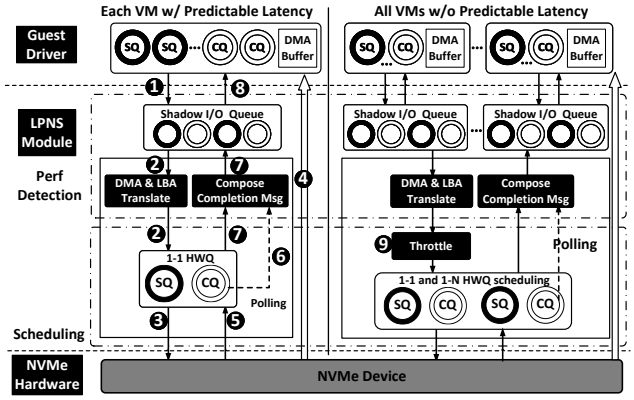
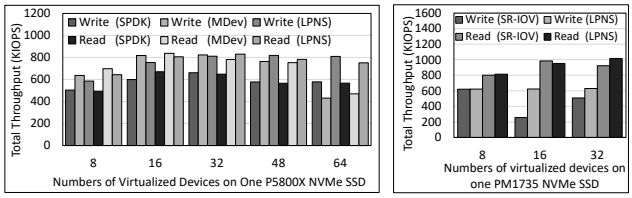


Figure 4: I/O work flow of LPNS virtualized storage.

The right part of Figure 4 shows that the IVMs get the HWQ resources from HWQ scheduling, and the polling thread can throttle the command distribution from the VQ to HWQ (where ❾ replaces the ❷). With the I/O path designs, the intensive workloads in these VMs will not hurt the latency of the VM with the predictable latency guarantees because of the device-level latency interference.

4.5 Discussion

Polling effectiveness. We discuss the polling effectiveness of LPNS by comparing the throughput of LPNS with MDev-NVMe, SPDK, and SR-IOV when multiple VMs share a P5800X/PM1735 SSD. Figure 5 demonstrates the total throughput where increasing numbers of VMs running an FIO workload with the "*numjobs*=1, *iodepth*=1" parameters share the same NVMe SSD. The results prove that LPNS can fully utilize the P5800X or PM1735 for multiple IVMs with only one polling thread and provide better scalability than MDev-NVMe, SPDK, and SR-IOV.



(a) Intel Optane P5800X (b) Samsung PM1735

Figure 5: Polling effectiveness for high throughput (SPDK and LPNS both use one polling thread shared by all VMs.)

We also run an FIO test with "*numjobs*=1 or 4, *iodepth*=1" on the host OS and inside a VM with MDev-NVMe, SPDK and SR-IOV. Figure 6 demonstrates how one dedicated polling thread can achieve promising and near-native average latency.

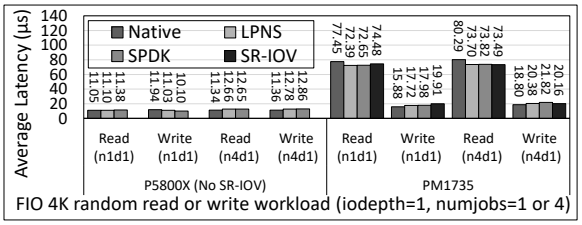


Figure 6: One dedicated polling thread for one SVM.

Performance overhead. In the self-feedback QoS control, each polling thread uses a two-phase array to achieve real-time performance data feedback for the SVMs. While the thread reads data from one phase of the array for computation, the probes record current I/O data into the other phase. It is lock-free, and only a writing-array operation is added into the origin command execution time. So the performance overhead of our self-feedback mechanism is negligible. Figure 5 and 6 prove that the active polling in LPNS can ensure no utilization overhead of the total IOPS and near-native idle latency performance for SVMs.

Resource overhead. (1) The choices of the performance detection are critical for resource overhead. The detection exacerbates kernel memory consumption because tens of thousand of commands can occupy hundreds of KBs of memory to store the performance data. Therefore, we cut the one-period detection into 10 ms intervals (much shorter than the scheduling period), we can calculate the average command latency for each interval, and finally sum up the results of a whole

period so we can reduce kernel memory overhead. (2) The polling threads in LPNS can be adaptively turned into an idle status when there are no I/O operations generated from the guest machines in a recent 500 ms. When performance detection finds there are burst I/O operations, LPNS will turn on the polling thread immediately. This helps reduce the CPU overhead of polling.

Limitation. The deterministic network calculus of LPNS needs tunning when cloud workloads use different NVMe SSDs or block sizes instead of the most commonly-used 4K. Also, for the SSDs whose random write IOPS is much lower than random read, the deterministic network calculus will be more strict to provide latency bound, and LPNS may sacrifice more throughput performance than when running LPNS on SSDs with equivalent random read and write performance, such as Intel Optane NVMe SSDs.

5 Evaluation

In this section, we evaluate LPNS on different NVMe SSDs and compare LPNS with several famous related works.

Firstly, we want to compare LPNS with mainstream NVMe virtualization solutions. We start with the following NVMe virtualization mechanisms: MDev-NVMe, VirtIO, SPDK with *vhost-blk* interfaces, and SR-IOV [2].

We also compare LPNS with state-of-the-art QoS control systems, MQFQ, D2FQ, and K2. Specifically, (1) we build a K2 kernel module with its source code [68]; (2) We implement MQFQ with 1125 Lines of Code according to the description in [21] since the original source [20] is no longer accessible. (3) We modify 19 lines of the D2FQ source codes [62] for bug fixing according to the description in [71].

5.1 Experiment Setup

Hardware configuration. We evaluate LPNS on two servers. One server has two 20-core Intel Xeon Gold 6248 CPUs (2.5GHz), 384GB DDR4 memory, and one 400GB Optane P5800X SSD [24]. Another server has two 20-core Intel Xeon Gold 6230 CPUs (2.1GHz), 128GB DDR4 memory, and one 1.6TB SR-IOV-capable Samsung PM1735 SSD [58]. The parameters Ω we choose for the P5800X and PM1735 are 190 and 100 (the choices are discussed in §5.5.)

System configuration. We implement LPNS based on Linux kernel 5.0.0. The two host servers run a Ubuntu 18.04.3 LTS 64bit OS and boot VMs with the same OS image version based on KVM/QEMU. There are different numbers of SVMs and IVMs in micro and real-world benchmarks. Each VM has 4 VCPUs, 4GB memory, 40GB virtual NVMe storage, and 4VQs equal to the number of VCPUs. Each virtual storage is created on a logical partition of the SSD, and the VM uses the original NVMe driver of Linux. In all experiments, the total number of HWQs used for virtualization is less than the total number of VQs to mimic a resource shortage scenario in the real-world cloud environments.

Workload configuration. The micro-workloads are generated by FIO [26], which is widely used in both industry and research. The FIO version is 3.1, and *libaio* is selected as the default I/O engine. We set the I/O mode as Direct I/O. We set the block size of random read/write as 4K. In application benchmarks, we first replay the webuser service of open-sourced production systems at Florida International University (FIU) [6]. We also use YCSB [8] as another application benchmark and choose RocksDB [3] to test the I/O performance of K-V store. The YCSB version is 0.17.0, and we use the embedded RocksDB database of YCSB [2].

5.2 Micro Benchmarks

In the micro benchmarks, we let one SVM to share a P5800X SSD with one or multiple IVMs on LPNS, MDev-NVMe, SPDK with *vhost-blk*, and VirtIO. We also compare LPNS with MDev-NVMe and SR-IOV on a PM1735. In each test case, the SVM runs the lightest FIO workloads in a general sense by setting both the two standard decisive FIO parameters “numjobs" and “iodepth" as 1 (n1d1). The other IVMs run throughput-insensitive workloads as competitors, and we let their workloads grow from as light as the SVM to heavy enough to reach the throughput limit of the SSD by increasing the “numjobs" and “iodepth" parameters. We observe the latency of the “n1d1" workload in the SVM when the SVM faces serious interference.

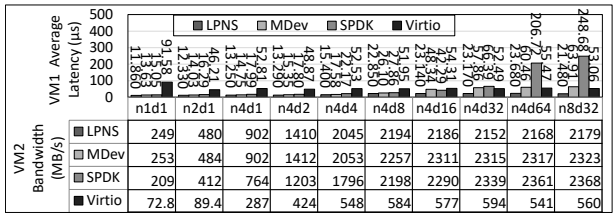
The micro benchmark results are demonstrated in Figure 7. Firstly, we let one SVM (VM1) and IVM (VM2) to share one underlying P5800X or PM1735 SSD and show the performance results in Figure 7a to 7d. The bar charts in the upper part of the sub-figures represent the latency performance of the FIO n1d1 workload in VM1, and the tables in the lower part are the throughput of the competitor workloads of VM2 in the same test cases. We also let one SVM (VM1) and multiple (2-7) IVMs to share the same SSD and demonstrate the latency of the VM1 workload in Figure 7e.

P5800X. We let the VM1 and the VM2 share one P5800X SSD, and the performance of 4K random read and write test cases are depicted in Figure 7a and 7b. Optane SSDs use 3D XPoint technology [17] (the most advanced storage medium), and their hardware controllers usually have stable storage service capability in latency and throughput, which is more friendly to latency-predictable systems. Because Optane SSDs also have similar random read and write performance, the results shown in Figure 7a and 7b have similar features.

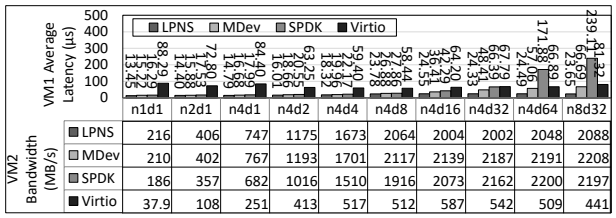
LPNS can bound the latency of VM1 workload at a low level (less than 25 μs) in both 4K random read and write cases. LPNS can only sacrifice the throughput of VM2 workload within 7.0% of MDev-NVMe and within 8.2% of SPDK when VM2 runs n464 or n832 cases, and in most cases, the sacrifices

[2]The hardware/software co-designed FVM [37] and LeapIO [41] are not open-sourced and available.

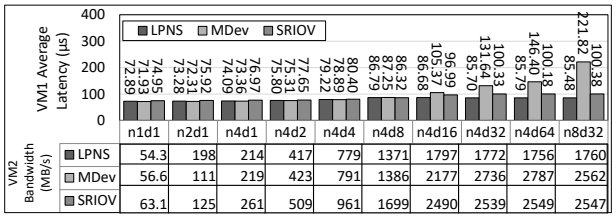
are less than 1%. Moreover, LPNS successfully eliminates the device-level interference observed in Figure 2.



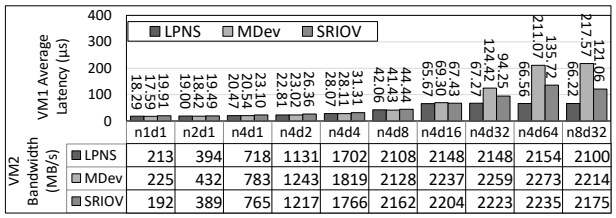
(a) One SVM & One IVM, 4K random read (P5800X).



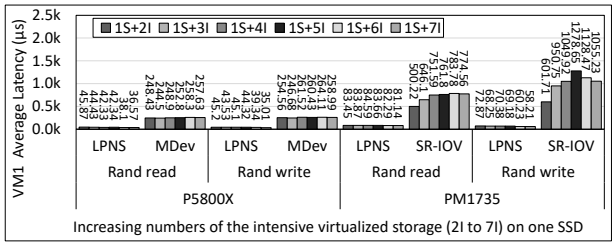
(b) One SVM & One IVM, 4K random write (P5800X).



(c) One SVM & One IVM, 4K random read (PM1735).



(d) One SVM & One IVM, 4K random write (PM1735).



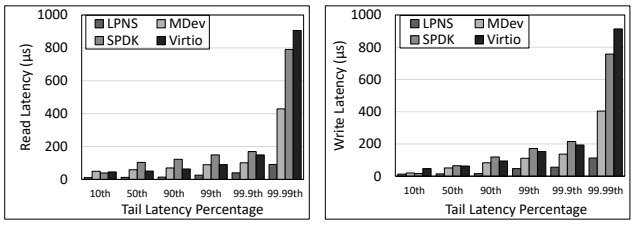
(e) One SVM & Multiple IVMs (P5800X & PM1735)

Figure 7: The average latency of LPNS compared with the other NVMe virtualization on P5800X and PM1735.

In MDev-NVMe and SPDK, the increasing throughput of the competitor workload in VM2 will lead to latency deterioration of the VM1 workload. For example, when using n4d64 or n8d32 parameters, LPNS can reach up to **11.57/2.98**× latency optimization over SPDK/MDev-NVMe. The main reason is that their polling threads cannot provide strong performance isolation between different virtualized devices to overcome latency interference. VirtIO can obtain stable average latency and better latency bound for VM1 than SPDK, but the VM1 latency and VM2 throughput are not comparable with LPNS because VirtIO is generic virtualization for block devices and not optimized for NVMe.

Figure 7e shows that LPNS can provide latency-predictable QoS with promising scalability for the SVM when we increase the number of IVMs from 2 to 7. The latency results of VM1 are very stable and low, and bounded by **50** μs, which can achieve up to 7.40× latency optimization of MDev-NVMe.

Figure 8 demonstrate the tail latency of the n8d32 test cases in Figure 7a and 7b. LPNS can reach up to 3.84/6.70× optimization of 99.9th/99.99th 4K random read tail latency of SPDK and 3.73/9.96× optimization of 99.9th/99.99th 4K random write tail latency of SPDK.



(a) Random Read Tail Latency (b) Random Write Tail Latency

Figure 8: Tail Latency of micro benchmarks on P5800X .

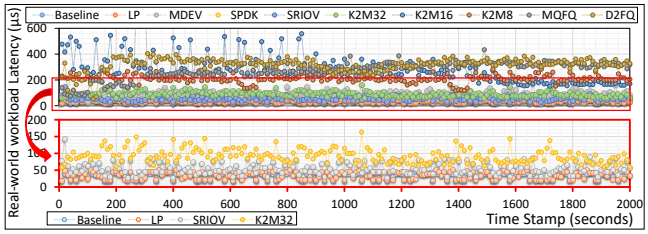
PM1735. We let the VM1 and the VM2 share one PM1735 SSD, and the performance results are shown in Figure 7c and 7d. These results show that LPNS can provide better latency than MDev-NVMe and SR-IOV in all cases and achieve up to 3.27× latency optimization. Moreover, when we increase the number of IVMs from 2 to 7 on PM1735, the IVMs cause serious latency interference on the VM1 when using SR-IOV. LPNS can bound the VM1 workload's latency under **90**μs, with up to **18.72**× latency optimization over SR-IOV. To be mentioned, the throughput performance loss of IVMs on PM1735 is worse than P5800X, which is less than 7.07% in the 4K random write cases, but up to 31.11% in the 4K random read cases. The main reason is that the random read service capability of the PM1735 controller is much better than the random write, and LPNS must choose a more conservative configuration to ensure latency-predictable QoS but incur more throughput loss.

In general, LPNS can perform better in both latency bound and low throughput loss than VirtIO. LPNS can provide ultra-low latency and ensure latency-predictable QoS compared to MDev-NVMe, SPDK, and SR-IOV.

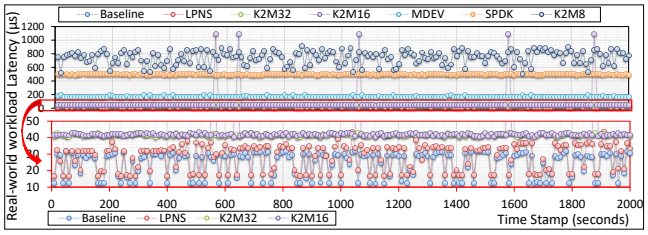
5.3 Real-world I/O trace Replay

In the real-world I/O trace replay, we run a *webuser* workload in the SVM(s) and let the SVM share one Optane P5800X and PM1735 SSD with one or multiple IVMs running intensive workloads (FIO 4K random write with *numjobs=4* and *iodepth=32*). The raw data of the latency-sensitive *webuser* server comes from [6]. We extract the throughput and the proportion of reading and writing operations from the raw data.

The running time of each *webuser* test case is 2000 seconds, and we report latency results every 10 seconds.



(a) Samsung PM1735.



(b) Intel Optane P5800X.

Figure 9: The latency of real-workload *webuser* on LPNS, MDev-NVMe, SPDK, MQFQ, D2FQ, and K2 with 32, 16, and 8 *max_inflight* when 4 VMs share one NVMe SSD (VM1 is an SVM and VM2-4 are IVMs), compared with a baseline that only VM1 monopolizes the NVMe SSD. The lower-part charts in the figures are the enlargement of the data from the red boxes of the upper-part charts.

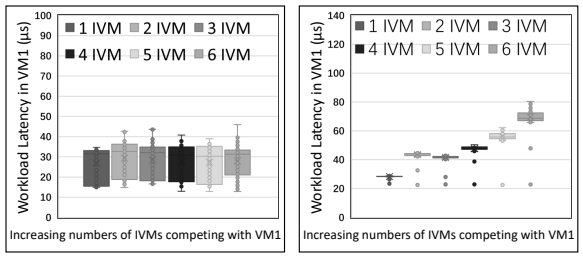
We choose MDev-NVMe and SPDK with *vhost-blk* as the representative of the state-of-the-art virtualization, and we design eight groups of experiments with different numbers of SVM (s) and IVM(s), including "1SVM & 1IVM" to "1SVM & 6IVM", "2SVM & 2IVM", and "3SVM & 1IVM".

We choose K2, MQFQ, and D2FQ for comparison. Since LPNS, MDev-NVMe, and SPDK can prove near-native latency for each virtualized storage and K2, MQFQ, and D2FQ are not designed for NVMe virtualization, we deduce that this latency performance comparison are typical and persuasive. We build the K2 module and *insmod k2-scheduler* into the original 4.15.0-175-generic kernel of the public Ubuntu 18.04 system. We implement MQFQ and D2FQ in a Linux 5.3.10 kernel. We run the same real-world *webuser* workloads and intensive FIO random write workloads on the host OS, just like inside the SVM(s) and IVM(s). We use the `ionice` [10] commands to give the *webuser* a high priority for latency QoS control on K2, MQFQ, and D2FQ scheduler. Specifically, we configure the *max_inflight* parameters of K2 as 32, 16, and 8, and the parameters of MQFQ and D2FQ are set as default values according to the papers.

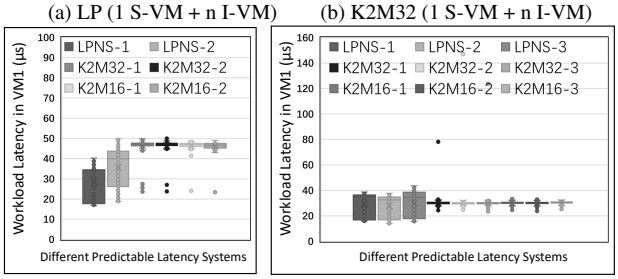
Latency fluctuation. We firstly choose the "1 SVM & 3 IVMs" cases on PM1735 and P5800X as a typical example of all the cases to show the overall latency control effects of the related works compared with a baseline where only VM1 running the *webuser* monopolizes the entire SSD. We demonstrate the latency results of the *webuser* in Figure 9.

On both PM1735 and P5800X SSD, the latency of LPNS can nearly coincide with the baseline fluctuation line, and it successfully ensures the *webuser*'s latency-predictable QoS. SR-IOV and K2 can also provide better latency performance than MQFQ, D2FQ, and SPDK because the main purpose of MQFQ and D2FQ is to reach a fair queue scheduling and SPDK does not involve reliable performance isolation. On P5800X, K2 with *max_inflight=32* can successfully bound the latency within **50**μs. However, SR-IOV and K2 cannot bound latency within 50μs level as LPNS does on PM1735.

Latency interference elimination. We count the latency distributions of the eight groups of LPNS and K2 (providing better latency bound effect in the example case) in Figure 10 with box charts.



(a) LP (1 S-VM + n I-VM) (b) K2M32 (1 S-VM + n I-VM)



(c) 2 S-VM+2 I-VM (d) 3 S-VM+1 I-VM

Figure 10: The latency distribution results of test cases where different numbers of SVMs and IVMs share one P5800X SSD on the LPNS system or on a system using K2.

In general, LPNS can bound the latency by 50 μs in all the test cases. K2 can bound the latency by 50 μs when there are small numbers (1∼4) of IVMs. However, the outliers of latency results in the box chart prove that the latency of VM1 *webuser* workload grows over 50 μs on K2 with *max_inflight=32* when there are more than 5 IVM as competitors. When there are more SVM but less IVM (for example, Figure 10c and 10d), K2 with *max_inflight=32 or 16* can effectively bound the latency by 50 μs just as LPNS does.

Throughput sacrifice. Figure 11 demonstrates the total throughput of the IVM(s) that compete for resources with the SVM in the "1SVM & 1IVM" to "1SVM & 6IVM" cases. We choose the maximum throughput of the P5800X as the baseline. LPNS throughput loss to MDev-NVMe is 18.46% in average and less than 19.88%. Figure 11 also proves that LPNS can reach latency-predictable QoS with less throughput loss of the IVMs than K2 when multiple VMs share the one SSD. For example, when there are 6 IVMs (or 6 native inten-

sive workloads), LPNS can increase up to **1.45** × additional total throughput over K2 with *max_inflight=32* (equivalent to **47.66% of the maximum throughput** of the P5800X SSD.)

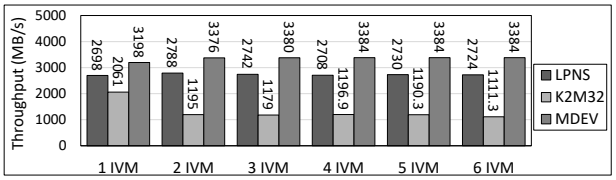


Figure 11: The total throughput of the one to six IVMs (or native workloads) on LPNS and K2 (*max_inflight=32*) in Figure 10a and 10b (MDev-NVMe as the baseline).

We deduce that LPNS will be an excellent choice to achieve their latency-predictable QoS with less sacrifice of the SSD maximum throughput when cloud vendors want to increase the scalability of virtualized devices on each single SSD with heavier over-subscription of storage resources.

5.4 YCSB Key-Value Store on RocksDB

We use the YCSB to generate K-V store workloads on the RocksDB databases in one SVM, and use several IVMs running intensive FIO random read/write workloads to generate serious interference and demonstrate that LPNS can provide latency-predictability for K-V store applications. Specifically, we build an `ext4` [1] file system on the virtualized NVMe storage device in the SVM to run the RocksDB database. We run the generic configuration of YCSB from the `workloada` to `workloadf`, which uses Zipfian [18] distributions. We set up 20M requests on 4GB database by enlarging the "recordcount" and "operationcount".

Figure 12 demonstrates the average latency performance results of YCSB benchmarks on the RocksDB databases, including the baseline where the benchmark monopolizes the entire P5800X SSD, or using MDev to support 1SVM +3IVM, or using LPNS to support 1SVM +3IVM. In the six YCSB tests, LPNS can efficiently reduce the latency interference from the IVMs and provide promising average latency performance for the Read, Update, Insert, or Scan operations. For example, the YCSB-A and YCSB-B workloads are identical mix Read and Update operations, which will bring challenges to the latency QoS control systems. The results prove that LPNS can achieve up to **7.41** × latency optimization of MDev-NVMe in the YCSB-B workloads. In YCSB-C, E, and F, MDev-NVMe and LPNS can provide latency similar to the monopolized baselines. We infer the reasons: YCSB-C is a 100% read case, which is very friendly to the cache systems, so it is not seriously interfered by the device-level congestion; YCSB-E and YCSB-F are more performance-critical about the computation ability of the K-V store databases, so the storage system is not a performance bottleneck in these cases.

In Table 3, we count the tail latency results of YCSB-A and YCSB-B benchmarks of MDev-NVMe and LPNS from Figure 12. From these results, we can find that LPNS can

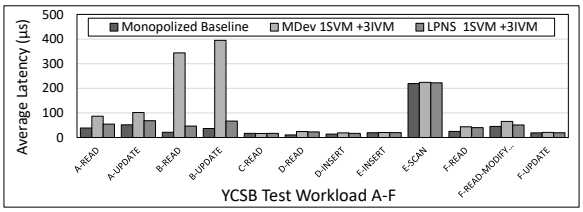


Figure 12: The average latency of YCSB A to F workloads.

achieve up to 4.44 × optimization of the 95th tail latency and up to 4.27 × optimization of the 99th tail latency compared with MDev-NVMe, which can provide the most advanced tail latency among the traditional NVMe virtualization in the Section 5.2 experiments.

Table 3: YCSB tail latency in YCSB-A and YCSB-B

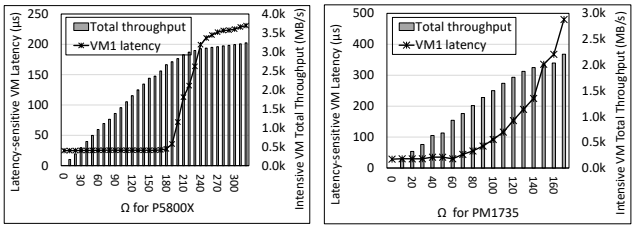
		Read		Update	
		MDev	LPNS	MDev	LPNS
YCSB-A	P95 (us)	419	242	453	260
	P99 (us)	896	366	934	399
YCSB-B	P95 (us)	907	204	961	225
	P99 (us)	969	317	1060	358

With these experimental results, we imply that LPNS can provide latency-predictable QoS for the latency-sensitive applications such as the K-V store databases in cloud services.

5.5 Predictability Trade-off

We discuss the trade-off between latency and throughput to guide the choice of the proper predicted value, which refers to the value of the I/O command scheduling parameter Ω in LPNS. The Ω value is related with the hardware SSD and can be flexibly determined by the cloud storage administrators to make an optimal alternative according to the hardware and workload scenarios. We observe the latency performance of SVM and the total throughput of IVMs to evaluate the effect of different Ω values and choose the proper Ω.

Figure 13 plots the latency of VM1 (SVM) and total throughput of IVMs at different Ω values on the P5800X and PM1735 SSDs. We use the dot lines to indicate the latency performance and bars to indicate the total throughput performance.



(a) Intel Optane P5800X. (b) Samsung PM1735.

Figure 13: Performance trade-off for different SSDs.

On P5800X, the R is 800K IOPS (0.8 iopμs) and L_h of the workload is 11.05 μs (measured when there is only one workload that monopolizes the SSD). When we choose Ω as 10, the latency can be bounded within 23.55μs according to Eq. 5, which matches the results in Figure 13a.

However, when Ω is smaller than 160, keeping decreasing Ω can not significantly improve the latency, but the throughput will continue to decline. When Ω is larger than 240, keeping increasing Ω is also uneconomic because the throughput is close to the ceiling while the latency is still growing. So an appropriate Ω value must be between 160 and 240, depending on the trade-off between the latency requirements and the acceptability of bandwidth degradation. We choose **190** on P5800X in all our experiments to bound the latency in micro and application benchmarks and get a balance between the latency bound (**50**μs) and the throughput sacrifice. With these configurations, we use Figure 14 to prove that LPNS can successfully bound the latency of the workload in Figure 1.

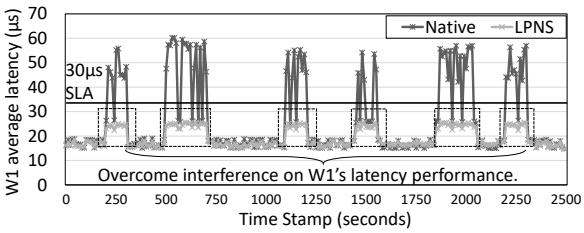


Figure 14: Latency bounded within 30μs without SLA miss.

Compared with P5800X, it is more difficult to reach a promising balance between latency and throughput on the PM1735. The Ω parameter when Ω is smaller than 60, keeping decreasing Ω can not significantly improve the predictable latency but only decline the throughput. And when we choose larger Ω, the latency grows faster than on P5800X. We finally choose **100** as the Ω to provide an upper bound of latency at **100**μs for the latency-sensitive workloads on PM1735. However, it will incur more serious throughput loss than using LPNS on P5800X. And this will be one of our future work to reach a better balance of latency prediction and throughput by upgrading our design and implementation.

6 Related Work

Local NVMe virtualization. The state-of-art local NVMe virtualization mechanisms include para-virtualization *VirtIO* [57], a userspace NVMe driver *VFIO* [70], and SPDK [25], MDev-NVMe [55], FinNVMe [54], FVM [37]. *VirtIO* is an I/O para-virtualization framework, which provides an abstraction of a set of common simulation devices suffers from the poor performance of virtualized devices. Fam Zheng [73] used *VFIO* to implement the NVMe driver to work with the modified user-space NVMe driver in Qemu. The SPDK is a userspace and lockless NVMe driver that provides an efficient and scalable interface to access various storage devices. MDev-NVMe uses mediated pass-through and the active polling mode to achieve high I/O performance. FVM implements a storage virtualization layer on an FPGA card to offload virtualization overhead, and FVM can ensure high throughput but incurs about 25% latency overhead over native performance.

Storage QoS. Many researches concentrate on NVMe resource sharing and scheduling by modifying the device controller or host software stack [4, 11, 19, 32, 39, 42, 43, 56, 60, 72]. FIOS [53] achieves the fairness of resource sharing through the per-task timeslices. AutoSSD [31] employs a self-management mechanism to schedule device-internal background jobs to prevent the SSD from falling into a critical condition that causes long tail latency. PartFTL [49] splits flash storage into separate read and write sets to ensure writes never block reads. FLIN [66] is a lightweight transaction scheduler using a three-stage scheduling algorithm to provide fairness, implemented within the SSD controller firmware. For performance isolation [15, 22, 30, 33, 36, 50, 61, 64] and QoS, workload-aware budget compensation (WA-BC) [29] provides a device-level scheduler with SR-IOV for performance isolation and fairness among multiple VMs by penalizing noisy neighbors. Differentiated Storage Services [47] propose an I/O classification architecture to close the gap between computer systems and storage systems, and improve end-to-end performance, reliability, and security of storage. More recent schedulers have evolved to guarantee isolation at OS-level without modification to the hardware. K2 [48] is a lightweight and device-agnostic I/O scheduler for Linux targeting NVMe-attached storage. Multi-Queue Fair Queueing (MQFQ) [21] is a fair and work-conserving scheduler for multi-queue systems. D2FQ [71] uses the device-side scheduling feature (NVMe WRR) to reach low-CPU-overhead fair queue scheduling.

7 Conclusion

In this paper, we propose LPNS, a latency-predictable NVMe virtualization mechanism for local cloud storage. Based on the mediated pass-through mechanism, LPNS retains high virtualization performance with I/O queue and command scheduling. To prove latency-predictable QoS, we model LPNS as a deterministic queuing system and deduce the latency upper bound referring to the deterministic network calculus. The evaluations demonstrate that LPNS can achieve the goal of latency predictability and prove to be an efficient cloud storage virtualization mechanism. In our future work, we will also improve the balance between latency prediction and throughput on more types of NVMe SSDs, and we are devoted to integrating LPNS into a hardware/software co-designed solution to offload overhead and improve scheduling efficiency.

Acknowledgments

We sincerely thank our shepherd, Youyou Lu, and the anonymous reviewers for their suggestions and help to improve the paper. This work was supported in part by National Key Research & Development Program of China (No. 2022YFB4500103), the Programs for NSFC (No. 62032008), the Resnics Project, and STCSM (No. 20510712400). The corresponding author is Jianguo Yao.

References

[1] Ext4 filesystem. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt/.

[2] how to run ycsb on rocksdb. https://github.com/brianfrankcooper/YCSB/tree/master/rocksdb/.

[3] A persistent key-value store for fast storage environments. https://rocksdb.org/.

[4] Sungyong Ahn, Kwanghyun La, and Jihong Kim. Improving I/O resource sharing of linux cgroup for nvme ssds on multi-core systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.

[5] Marc Boyer, Euriell Le Corronc, and Anne Bouillard. *Deterministic network calculus: From theory to practical implementation*. John Wiley & Sons, 2018.

[6] Camelab. Flash-based block traces. http://trace.camelab.org/2016/03/01/flash.html.

[7] Yiquan Chen, Jiexiong Xu, Chengkun Wei, Yijing Wang, Xin Yuan, Yangming Zhang, Xulin Yu, Yi Chen, Zeke Wang, Shuibing He, et al. Bm-store: A transparent and high-performance local storage architecture for bare-metal clouds enabling large-scale deployment. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1031–1044. IEEE, 2023.

[8] 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.

[9] Intel Cooperation. AHCI specification for Serial ATA. https://www.intel.com/content/www/us/en/io/serial-ata/ahci.html.

[10] die.net. ionice(1) - Linux man page. https://linux.die.net/man/1/ionice.

[11] Adam Ji Dou, Song Lin, and Vana Kalogeraki. Real-time querying of historical data in flash-equipped sensor devices. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, pages 335–344. IEEE Computer Society, 2008.

[12] Nathan Farrington and Alexey Andreyev. Facebook's data center network architecture. In *Proceedings of the 2013 Optical Interconnects Conference, OI 2013, Santa Fe, NM, USA, May 5-8, 2013*, pages 49–50, 2013.

[13] Fabien Geyer and Steffen Bondorf. Deeptma: Predicting effective contention models for network calculus using graph neural networks. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1009–1017. IEEE, 2019.

[14] PCIe Special Interest Group. Welcome to pci-sig. https://pcisig.com/, 2020.

[15] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: proportional allocation of resources for distributed storage access. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pages 85–98. USENIX, 2009.

[16] Ajay Gulati, Arif Merchant, and Peter J. Varman. mclock: Handling throughput variability for hypervisor IO scheduling. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 437–450. USENIX Association, 2010.

[17] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[18] S Haitun. Stationary scientometric distributions: Part iii. the role of the zipf distribution. *Scientometrics*, 4(3):181–194, 1982.

[19] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020.

[20] hedayati. . https://github.com/hedayati/mqfq.

[21] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-queue fair queuing. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 301–314. USENIX Association, 2019.

[22] 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 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 375–390. USENIX Association, 2017.

[23] Khoa Huynh. Exploiting the latest kvm features for optimized virtualized enterprise storage performance. *CloudOpen, North America*, 2013.

[24] Intel. Intel® Optane™ SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201861/intel-optane-ssd-dc-p5800x-series- 400gb-2-5in-pcie-x4-3d-xpoint.html.

[25] Intel. Storage performance development kit. http://www.spdk.io/.

[26] Axboe Jens. Fio: Flexible io tester. https://github.com/axboe/fio.

[27] Neo Jia. VFIO Mediated devices. https://www.kernel.org/doc/Documentation/vfio-mediated-device.txt.

[28] Yichao Jin, Yonggang Wen, and Qinghua Chen. Energy efficiency and server virtualization in data centers: An empirical investigation. In *Proceedings of the 2012 Proceedings IEEE INFOCOM Workshops, Orlando, FL, USA, March 25-30, 2012*, pages 133–138. IEEE, 2012.

[29] Byunghei Jun and Dongkun Shin. Workload-aware budget compensation scheduling for nvme solid state drives. In *Proceedings of the IEEE Non-Volatile Memory System and Applications Symposium, NVMSA 2015, Hong Kong, China, August 19-21, 2015*, pages 1–6. IEEE, 2015.

[30] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage '14, Philadelphia, PA, USA, June 17-18, 2014*. USENIX Association, 2014.

[31] Bryan Suk Kim, Hyun Suk Yang, and Sang Lyul Min. Autossd: an autonomic SSD architecture. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 677–690. USENIX Association, 2018.

[32] Jae-Hong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware I/O management for solid state disks (ssds). *IEEE Trans. Computers*, 61(5):636–649, 2012.

[33] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO complying ssds through OPS isolation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 183–189. USENIX Association, 2015.

[34] Jungkil Kim, Sungyong Ahn, Kwanghyun La, and Wooseok Chang. Improving i/o performance of nvme ssd on virtual machines. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1852–1857. ACM, 2016.

[35] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 29. ACM, 2016.

[36] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 345–359. ACM, 2017.

[37] 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 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 955–971. USENIX Association, 2020.

[38] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[39] Minkyeong Lee, Donghyun Kang, Minho Lee, and Young Ik Eom. Improving read performance by isolating multiple queues in nvme ssds. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication, IMCOM 2017, Beppu, Japan, January 5-7, 2017*, page 36. ACM, 2017.

[40] Chengzhi Li, Almut Burchard, and Jörg Liebeherr. A network calculus with effective bandwidth. *IEEE/ACM Transactions on Networking*, 15(6):1442–1453, 2007.

[41] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 591–605, New York, NY, USA, 2020.

[42] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. Loda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 263–279, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. Rail: Predictable, low tail latency for nvme flash. 18(1), jan 2022.

[44] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 301–312. IEEE Computer Society, 2014.

[45] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible colocations. In *Proceedings of the 44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, pages 248–259. ACM, 2011.

[46] Mellanox. Mellanox nvme snap™ | mellanox technologies. https://www.mellanox.com/products/software/nvme-snap, 2020.

[47] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 57–70, New York, NY, USA, 2011. Association for Computing Machinery.

[48] Till Miemietz, Hannes Weisbach, Michael Roitzsch, and Hermann Härtig. K2: work-constraining scheduling of nvme-attached storage. In *Proceedings of the IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 56–68. IEEE, 2019.

[49] Katherine Missimer and Richard West. Partitioned real-time NAND flash storage. In *Proceedings of the 2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 185–195. IEEE Computer Society, 2018.

[50] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 17–33. USENIX Association, 2017.

[51] Nvmexpress. Nvm express specification. http://www.nvmexpress.org/specifications/.

[52] 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 ACM SIGARCH Computer Architecture News*, volume 42, pages 471–484. ACM, 2014.

[53] Stan Park and Kai Shen. FIOS: a fair, efficient flash I/O scheduler. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 13. USENIX Association, 2012.

[54] Bo Peng, Ming Yang, Jianguo Yao, and Haibing Guan. A throughput-oriented nvme storage virtualization with workload-aware management. *IEEE Transactions on Computers*, 70(12):2112–2124, 2020.

[55] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. Mdev-nvme: A nvme storage virtualization solution with mediated pass-through. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 665–676. USENIX Association, 2018.

[56] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. Real-time flash translation layer for NAND flash memory storage systems. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, Beijing, China, April 16-19, 2012*, pages 35–44. IEEE Computer Society, 2012.

[57] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[58] Samsung. Enterprise ssd | ssd | samsung semiconductor. https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1733-pm1735/mzplj1t6hbjr-00007/.

[59] Jens B Schmitt, Frank A Zdarsky, and Markus Fidler. Delay bounds under arbitrary multiplexing: When network calculus leaves you in the lurch... In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 1669–1677. IEEE, 2008.

[60] Kai Shen and Stan Park. Flashfq: A fair queueing I/O scheduler for flash-based ssds. In *Proceedings of the 2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 67–78. USENIX Association, 2013.

[61] 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 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 349–362. USENIX Association, 2012.

[62] skkucsl. . https://github.com/skkucsl/d2fq.

[63] Yongseok Son, Hyuck Han, and Heon Young Yeom. Optimizing file systems for fast storage devices. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 8. ACM, 2015.

[64] Xiang Song, Jian Yang, and Haibo Chen. Architecting flash-based solid-state drive for high-performance I/O virtualization. *IEEE Comput. Archit. Lett.*, 13(2):61–64, 2014.

[65] David Starobinski, Mark Karpovsky, and Lev A Zakrevski. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Transactions on Networking*, 11(3):411–421, 2003.

[66] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri-Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. FLIN: enabling fairness and enhancing performance in modern nvme solid state drives. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 397–410. IEEE Computer Society, 2018.

[67] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *Proceedings of 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, 2014.

[68] tmiemietz. TUS-OS/k2-scheduler. https://github.com/TUD-OS/k2-scheduler.

[69] Kai Wang, Florin Ciucu, Chuang Lin, and Steven H Low. A stochastic power network calculus for integrating renewable energy sources into the power grid. *IEEE Journal on Selected Areas in Communications*, 30(6):1037–1048, 2012.

[70] Alex Williamson. Vfio: A user's perspective. https://www.linux-kvm.org/images/b/b4/2012-forum-VFIO.pdf.

[71] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. D2FQ: Device-Direct fair queueing for NVMe SSDs. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 403–415. USENIX Association, 2021.

[72] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 477–492. USENIX Association, 2018.

[73] Fam Zheng. Userspace nvme driver in qemu. https://events.static.linuxfound.org/sites/events/files/slides/Userspace%20NVMe%20driver%20in%20QEMU%20-%20Fam%20Zheng_0.pdf.

[74] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

P²CACHE: Exploring Tiered Memory for In-Kernel File Systems Caching

Zhen Lin*, Lingfeng Xiang†, Jia Rao†, Hui Lu*
*Binghamton University, †The University of Texas at Arlington

Abstract

Fast, byte-addressable persistent memory (PM) is becoming a reality in products. However, porting legacy kernel file systems to fully support PM requires substantial effort and encounters the challenge of bridging the gap between block-based access granularity and byte-addressability. Moreover, new PM-specific file systems remain far from production-ready, preventing them from being widely used. In this paper, we propose **P²CACHE**, a novel in-kernel caching mechanism to explore how legacy kernel file systems can effectively evolve in the face of fast, byte-addressable PM. P²CACHE exploits a read/write-distinguishable memory hierarchy upon a tiered memory system involving both PM and DRAM. P²CACHE leverages PM to serve all write requests for instant data durability and strong crash consistency while using DRAM to serve most read I/Os for high I/O performance. Further, P²CACHE employs a simple yet effective synchronization model between PM and DRAM by leveraging device-level parallelism. Our evaluation shows that P²CACHE can significantly increase the performance of legacy kernel file systems – e.g., by 200x for RocksDB on Ext4 – meanwhile equipping them with instant data durability and strong crash consistency, similar to PM-specialized file systems.

1 Introduction

Rapid changes in storage technologies, ranging from rotating hard disk drives (HDD) to NAND-based solid-state drives (SSD), Non-Volatile Memory Express (NVMe) [9], and Storage Class Memory (SCM) [10], play an essential role in driving the evolution of kernel file systems, such as Ext4 [35], Btrfs [37], and XFS [24]. Features have been continuously added to accommodate new and unique characteristics of storage devices. Examples include I/O schedulers designed for different types of storage media [21, 31, 39], concurrency and scalability support for high-speed storage on multi-core systems [17, 20], and the introduction of direct access mode (DAX) for byte-addressable SCM [5, 22].

However, the evolution of kernel file systems hits a plateau in light of emerging *fast, byte-addressable* storage [9, 10]. Kernel file systems are inherently built with the assumption of *slow, block-addressable* storage devices (e.g., HDD/SSD) sitting below. The layered in-kernel storage stack transforms I/O requests from applications to block operations for storage devices. As storage devices become faster, the overhead of the layered storage stack becomes more significant. For example, software contributes 50% of the read latency on a low latency (3 us) NVMe SSD [45]. The software overhead becomes even more dominant as storage devices become faster and closer to the CPU. Intel's Optane Persistent Memory [10] sitting on the memory bus incurs read latency as low as ∼170 ns [44].

To address this pressing challenge, some approaches tended to discard traditional kernel file systems. Indeed, a batch of new file systems [5, 18, 19, 23, 27, 28, 40, 42, 43] has been proposed – i.e., tailored for persistent memory (PM) [10] – and achieved high I/O bandwidth, low I/O latency, and strong crash consistency. Other approaches bypassed the kernel storage stack by exposing PM or low-latency SSDs directly to applications with userspace libraries [2] or file systems [32]. However, such new file systems and storage mechanisms may take a long time to mature and become production-ready – e.g., before having sufficient features and bug fixings.

We, instead, seek to answer the question: Can existing *well-tested, production-ready* kernel file systems effectively evolve to harness performance benefits and new characteristics of emerging storage technologies, achieving the same properties as those device-specialized file systems while requiring *no* application modifications and radical system redesign?

To answer this question, we gather the main insights from PM/SSD-specialized approaches that focus on minimizing system software overhead. First, maximizing the performance advantages of fast storage devices involves avoiding as much work as possible in the critical path. For example, approaches like SplitFS [27] and Strata [28] use a userspace library to handle data (and metadata) operations directly, which are then asynchronously processed by the kernel file system. Second, fast storage devices, along with a lightweight journaling/logging mechanism, can enable strong consistency with little overhead. For instance, NOVA [43] ensures that each file system update is synchronously persisted in an atomic manner.

Third, PM or modern NVMe SSD, with the advent of the high-speed CPU-to-device interconnect technology like Compute Express Link [4], provide a *memory-like, byte-addressable* interface [26], offering new design and optimization opportunities, including efficient handling of small writes for file system updates without write amplification.

These insights lead us to reap fast, byte-addressable storage for legacy kernel file systems with a novel in-kernel caching mechanism, **P²CACHE**. P²CACHE exploits a new *read/write-distinguishable memory hierarchy* within a tiered memory system involving both PM and DRAM. P²CACHE leverages fast PM to serve *all* write requests for instant data durability and strong crash consistency while using DRAM to serve *most* read I/Os for high I/O performance because DRAM's read performance remains significantly higher than PM.

P²CACHE first introduces a *persistent cache* located below the VFS layer. The persistent cache uses PM to quickly and synchronously persist/buffer file system metadata/data updates, guaranteeing instant data durability and strong crash consistency. The buffered operations are then asynchronously applied to underlying kernel file systems via the existing I/O interface, ensuring compatibility with kernel file systems. The persistent cache is built upon a lightweight operation log that captures minimal operations (i.e., file system updates from the VFS) and records them in a write-ahead log. It leverages PM's byte-addressability to efficiently persist metadata/data updates without costly, block-based Copy-on-Write (CoW) (commonly used in PM-based approaches [19,23,43]) – by decoupling the copy operation from the write operation, where a write operation (of any size) is synchronously recorded while the data copy is performed asynchronously.

P²CACHE further advances the *page cache* to serve most reads via faster but volatile DRAM. To allow the two caches – namely, the persistent cache and page cache – to work collaboratively and efficiently, P²CACHE leverages "device-level" parallelism. Specifically, we observe that the I/O latency of writing data to both PM and DRAM (at the same time) is almost the same as that of writing data to PM only, because the latency of the (extra) copy to DRAM is hidden (overlapped) by the *parallel-but-slower* PM write. This leads us to adopt a simple and effective *inclusive cache model*, where multiple copies of the same data are stored across the tiered memory, and the topmost layer (i.e., DRAM) always contains the latest version. The inclusive cache model simplifies the synchronization between the two caches: For writes, P²CACHE updates both caches; for reads, P²CACHE searches from the page cache, persistent cache, and underlying file systems sequentially until it first finds the data.

The benefits of P²CACHE are manifold: (1) P²CACHE requires *no* modifications to user applications, libraries, or kernel file systems while leveraging fast storage technologies to provide high I/O performance, high I/O concurrency, instant data durability, and strong consistency for legacy kernel file systems. Meanwhile, kernel file systems can still operate with their own (slow) storage devices (e.g., HDD/SSD). (2) P²CACHE does not provide complex file system functionalities (e.g., maintaining in-memory/on-disk data structures or disk block management). Instead, it focuses on efficiently persisting and buffering file system updates using PM as a persistent cache. It is extremely lightweight and enables legacy kernel file systems to achieve higher performance than PM-specialized file systems (e.g., NOVA [43]). (3) Unlike existing PM-based approaches that fully bypass DRAM (or page cache), P²CACHE leverages DRAM to maximize I/O performance – i.e., although PM has similar write latency as DRAM, there is a considerable latency gap for reads (e.g., 3x slower [44]). (4) Persistently buffering file system operations (i.e., metadata/data) enables new system optimizations. For example, P²CACHE accelerates the performance of a *cold-start* file system (e.g., after re-mounting or recovering from a system crash) by quickly re-building its in-memory cache (e.g., dentry cache) from the persistent cache.

We have implemented P²CACHE as a Linux kernel module interfacing with the VFS layer. Our evaluation with microbenchmarks and applications shows that the read/write-distinguishable memory hierarchy allows P²CACHE to significantly increase the performance of legacy file systems (e.g., Ext4) by up to 200x for RocksDB [1] while providing instant data durability and strong crash consistency. P²CACHE also achieves higher I/O performance than existing PM-specialized file systems, e.g., by up to 70% for RocksDB to NOVA [43].

2 Motivation

2.1 Fast Storage and Interconnect

Enabled by new storage technologies, such as 3D XPoint [12], NVMe SSDs over the PCIe bus achieve much higher bandwidth (e.g., 8 GB/s under the 70/30 mixed read/write case) and lower latency (e.g., as low as 3μs) [45] than before.

Further, byte-addressable persistent memory (PM) has been commercially available in a DIMM package on the memory bus, e.g., Intel Optane DC persistent memory [10]. PM allows programs to directly access data in non-volatile memory from the CPU using `load` and `store` instructions. PM offers approximately an order of magnitude higher capacity than DRAM (e.g., 8x capacity in Optane DIMMs) and within an order of magnitude performance of DRAM [44] (e.g., as low as 80 ns for write latency and 170 ns for read latency).

Despite Intel discontinuing its Optane product, the storage community is actively embracing high-speed CPU-to-device interconnect technologies [11], such as Compute Express Link (CXL) [4]. CXL provides a more general, unified interface to disaggregate various types of storage devices (e.g., DRAM, PM, and PCIe devices) directly to the CPU. CXL has the potential to offer a memory-like, byte-addressable alternative (i.e., via `load` and `store` instructions) to PCIe storage's block interface with minor modifications [26]. We envision that this trend will continue – storage devices will be increasingly faster with higher bandwidth and lower latency and offer byte-addressability via a memory-like interface. Although our

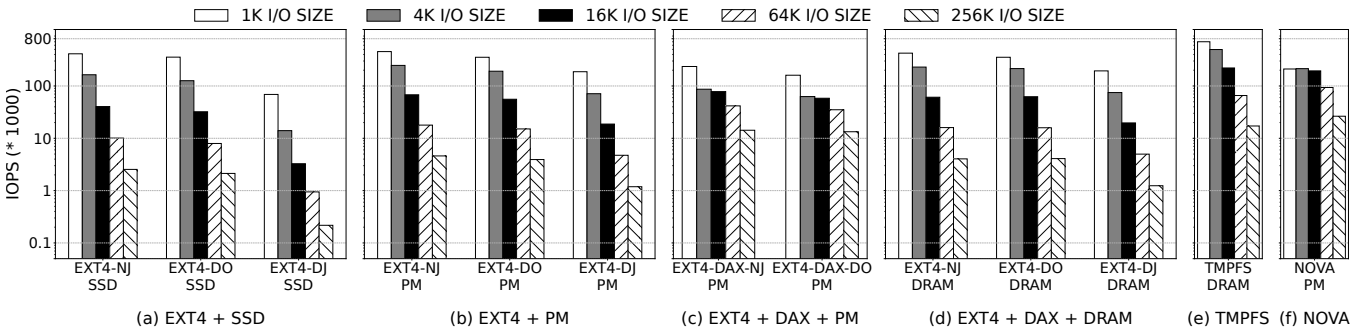


Figure 1: I/O performance comparisons for writes between cases with the combination of (1) distinct file systems: EXT4, EXT4-DAX, tmpfs, and NOVA; (2) journaling mode: no journal (NJ), data order (DO), and data journal (DJ); (3) storage medium: SSD, PM, and DRAM; and (4) various I/O sizes: 1 KB, 4 KB, 16 KB, 64 KB, and 256 KB.

work focuses on DIMM-based PM, it sheds light on bridging the gap between (1) byte-addressability, (2) a user-friendly and backward-compatible programming interface in PM and future CXL memory, and (3) the inherent differences between PM and traditional DRAM-based memory.

2.2 Kernel File Systems

Kernel file systems [7, 24, 29, 35, 37] have undergone continuous development and evolution, with the addition of features, bug fixes, and improvements in performance and reliability. For example, Ext4, the default general-purpose file system in most Linux distributions, was initially released in 2001.

The in-kernel storage stack converts I/O requests from applications into block operations that are persisted to storage devices through multiple software layers – i.e., the virtual file system (VFS), kernel file systems, generic block layers, journaling, and device drivers. While the layered design of the storage stack works well with slow underlying storage devices, such as HDD/SSD, it introduces nontrivial software overhead that becomes more pronounced as storage devices become faster. For example, in the case of low latency (3 us) NVMe SSD, software contributes 50% to read latency [45]. Our study on faster devices confirms this observation. *Observation 1: The in-kernel storage stack becomes the dominant storage bottleneck, rendering traditional kernel file systems unable to fully explore the potential of fast storage devices*. As depicted in Figure 1 (a), (b), and (d), Ext4 demonstrates only a marginal improvement in I/O performance (for writes) with PM (or even DRAM [1]) compared to SSD, despite the considerably faster speeds of PM/DRAM over SSD.

Virtual file system: Not all in-kernel storage layers contribute equally to the storage software overhead. The VFS, sitting atop the storage stack, incurs less than 10% of the overall software overhead [45]. Further, `tmpfs`, a lightweight kernel file system that works atop DRAM, achieves the highest performance (Figure 1(e)), indicating that a thin file system beneath the VFS can still leverage the benefits of fast storage media. On the other hand, the VFS implements key file system abstractions (e.g., `inodes` to represent metadata of on-disk files/directories) and functionalities (e.g., `pathname lookup` to locate a file/directory given a path name and `dcache` to cache such mappings in DRAM for quick lookup), commonly used by underlying file systems. *Observation 2: A storage layer (e.g., a persistent cache or page cache) – sitting below the VFS – can reuse VFS's rich functionality while being slim.*

Page cache: Because traditional disk accesses (e.g., HDD or SSD) are significantly slower than DRAM accesses, the operating system (OS) keeps frequently-accessed disk blocks in a dedicated region of DRAM, namely the *page cache*. The page cache reduces the number of disk accesses and speeds up I/O performance. However, as data updates are first applied to the page cache and later flushed to the storage, data modifications may not immediately reflect in the backing storage in case of sudden system crashes or power losses. This could cause an on-disk file system to enter an *inconsistent* state [34].

Journaling: To provide consistent and recoverable updates for metadata and/or data, kernel file systems often use *journaling* techniques. For example, Linux Ext4 employs the journaling block device version 2 (JBD2) [15], which implements a write-ahead log to record updates to a journal area before applying them to the corresponding file system locations. In the event of system failures, it replays the journal to restore the file system to a consistent state. JBD2 operates with three journaling modes: writeback, ordered, and data modes, providing trade-offs between performance and consistency. In writeback and ordered modes, only the metadata is journaled. However, the ordered mode (i.e., the default option) imposes an ordering requirement that data must be completed before the associated metadata is committed. This ordering constraint ensures stronger file system consistency than writeback, albeit with increased journaling latency [36]. On the other hand, the data mode journals both data and metadata, offering the highest level of consistency at the expense of the highest performance overhead.

Observation 3: Journaling places a serious impediment to the high performance of kernel file systems and can offset the performance benefits provided by the page cache. Figure 1 (a) and (b) show that in the ordered journal mode, the write performance under SSD and PM drops by ∼20% compared

[1] We used DRAM to emulate fast storage devices.

to the "no journal mode". The "data mode" substantially diminishes the write performance to only ∼10% of the no journal mode for SSD and ∼30% for PM. It is because (1) journaling requires writing metadata/data *twice*[2]: one to the log area and one to the file system location; though journaling is performed asynchronously, it can be frequently invoked by a background thread, competing for system resources with user applications. (2) JBD2 operates at the *block level* beneath the file system layer, recording all modified metadata/data blocks in a block unit within the journal area, even if only a small portion of the metadata/data blocks is modified (i.e., the small write case) – causing *write amplification*. (3) JBD2 journals all file system updates with a single kernel thread, limiting its scalability. In addition, the existing journaling scheme cannot ensure instant data durability. Instead, it relies on an explicit synchronous operation from users (e.g., `fsync()` or `fdatasync()`) for instant data durability.

2.3 Related Work

A large body of work has been recently proposed to exploit performance benefits and unique characteristics of fast storage technologies. We categorize them as two groups:

PM-specialized file systems: Many PM-specialized file systems have been studied [5, 18, 19, 23, 27, 28, 40, 42, 43]. They are tailored for the fast, byte-addressable PM to address challenges in write ordering and update atomicity, providing various levels of data/metadata consistency. For instance, BPFS [19] leveraged shadow paging for metadata/data consistency; PMFS [23] used journaling for metadata update atomicity, while performing in-place update for data (no atomicity and consistency); and NOVA [43] adopted a per-inode log-structured file system that offers synchronous persistence. Figure 1 (f) shows that NOVA achieves higher performance than Ext4 with the data journal mode (i.e., EXT4-DJ in Figure 1 (b)), while ensuring strong consistency. User-level PM-specialized file systems have been proposed to mitigate the overhead of system calls: Aerie [40] implemented a POSIX-like file system in userspace; Strata [28] appended updates to a userspace per-process log; and SplitFS [27] used a userspace library to handle data operations while still relying on the kernel-level file system to handle metadata operations.

Observation 4: the process of adapting existing storage systems to PM-specialized file systems remains in its early days. Making any of those projects production-ready needs substantial effort (and years) to achieve a combination of high performance, strong consistency, and comprehensive data protection. In addition, PM-specialized file systems unanimously bypass DRAM. While eliminating DRAM simplifies system design by allowing direct access to PM and avoiding synchronization complexity between PM and DRAM, it results in sub-optimal I/O performance. For example, the read latency of Intel Optane PM is *2x-3x* higher than that of DRAM, while the write bandwidth is only *1/6* [44]. Further compounding this situation, PM-specialized file systems usually trade I/O performance for write atomicity and consistency, for example, via log-structured file system techniques (cumbersome to reads) or shadow paging (causing write amplification) [19,43]. Figure 1 (f) shows that using Copy-on-Write (CoW) in NOVA to ensure data consistency incurs nontrivial overhead and proves detrimental to small writes (e.g., 1 KB).

[2]Although P^{2}CACHE also writes metadata and data twice, it has a very lightweight design with low overhead.

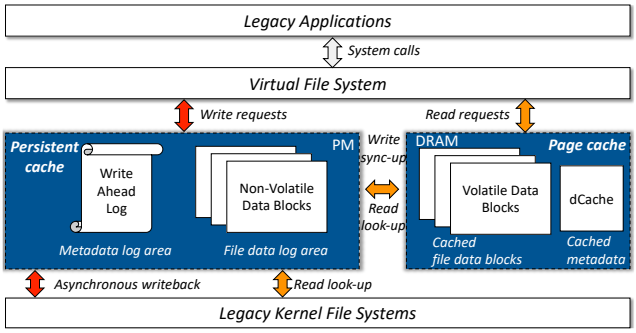


Figure 2: Overview of P^{2}CACHE.

PM-enhanced file systems: A straightforward approach for traditional block-based kernel file systems to adopt PM is to extend them with Direct Access (DAX), such as Ext4-DAX and XFS-DAX [5]. DAX-enabled kernel file systems bypass the page cache and perform reads/writes directly to PM. However, DAX-enabled file systems still operate at the block level and cannot explore PM's byte-addressability and its performance benefits. Figure 1 (c) shows that Ext4-DAX in the data-order mode achieves less than 50% of the performance of NOVA. Further, Ext4-DAX lacks support for the data journal mode and does not provide strong crash consistency.

Mostly related to P^{2}CACHE, efforts that explored the non-volatile nature of PM in building a cache layer have been conducted [30, 38]. They united either the journaling and caching [30] or the storage and caching [38] functionalities into a single cache layer. However, they are limited due to (1) like PM-specialized file systems, they eliminate DRAM despite its significant performance benefits for reads; (2) they only focus on caching data, while metadata operations remain on the slow path. Note that metadata can take up more than 50% of file system operations [3]; (3) they remain working at the block level (e.g., block-based CoW for data journaling or read-modify-write for partial writes), failing to leverage the byte-addressability of PM. *Observation 5: It is vital to have an OS caching approach that recognizes both data and metadata and distinct characteristics between PM and DRAM*.

3 P^{2}CACHE

We introduce P^{2}CACHE, a novel in-kernel caching mechanism. The goal of P^{2}CACHE is to enable the key properties of PM-specialized file systems for legacy kernel file systems, including instant data durability, strong consistency, high performance, and high concurrency while requiring *no* modifications to user applications, libraries, and kernel file systems. As illustrated in Figure 2, the key idea behind P^{2}CACHE is to

exploit a read/write-distinguishable memory hierarchy within the tiered PM/DRAM system, where P^{2}CACHE leverages fast PM to serve all write requests for instant data durability and strong crash consistency, while relying on DRAM to handle most read I/Os for high performance. P^{2}CACHE comprises two key kernel components: a persistent cache (Section 3.2) and a page cache (Section 3.3). Based on the observations in Section 2, P^{2}CACHE adopts the following key design choices.

3.1 Design Overview

A read/write-distinguishable memory hierarchy: We share the same observation as [41] that *the (modern) storage hierarchy is not a hierarchy* given the advent of fast, byte-addressable storage like PM. Unlike a traditional hierarchy where all I/O requests are first handled by the upper performance layer(s) and then consumed by the lower capacity layer(s), P^{2}CACHE distinguishes read and write operations in the PM/DRAM memory hierarchy with the goal to allow PM to handle all write requests while DRAM to serve most read I/Os because (1) P^{2}CACHE must persist each update in PM for instant data durability and strong crash consistency; and (2) DRAM's read performance is significantly higher than PM. To achieve this, P^{2}CACHE employs the following read/write strategies: (1) All writes are directed to PM, with a copy of the data modification also made in DRAM. It ensures that both PM and DRAM have the same data version. (2) Reads are first served from DRAM. If not found, P^{2}CACHE searches PM and underlying file systems. While P^{2}CACHE writes data to both PM and DRAM, it leverages *device-level parallelism* with *little* performance degradation – the latency of each data copy to DRAM is hidden by the parallel (slower) PM write.

A lightweight operation log: To harness the high performance of PM, P^{2}CACHE minimizes the operations in the critical path that involves PM. As shown in Figure 2, P^{2}CACHE's persistent cache resides at an early I/O layer, just below the VFS to leverage its general file system abstractions and functionalities while being slim. The role of the persistent cache is to capture all file system updates from the VFS (e.g., data overwrites/appends and metadata updates) and quickly, atomically persist them in an operation log stored in PM.

A PM-optimized logging mechanism: P^{2}CACHE's operation log in PM consists of two log areas: one for metadata updates and one for file data updates. First, P^{2}CACHE uses fixed-sized log entries (e.g., 64 bytes) to record metadata updates in the *metadata log area*. For data updates, P^{2}CACHE uses different strategies: (1) For *unaligned* overwrites (i.e., covering one or spanning across two *partial* data blocks), P^{2}CACHE directly appends the data to the end of its metadata update log entry for fast persistence. (2) For *aligned* overwrites (i.e., covering one or multiple contiguous full data blocks), P^{2}CACHE allocates free data blocks in the *file data log area* to store the data. (3) For *data appends*, P^{2}CACHE simply stores the appended data at the end of its data blocks in the file data log area (allocating additional blocks, if necessary). This approach ensures data consistency by never overwriting any old

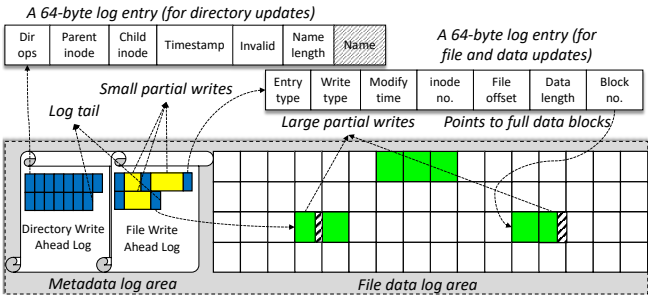


Figure 3: The layout of one CPU core's PM space.

data before the commit stage and avoiding the costly CoW operations commonly employed in other PM-based solutions.

Fast reads via in-DRAM indexes: While P^{2}CACHE's persistent cache benefits writes, it challenges reads. For example, a read operation may involve data scattered across various locations, and P^{2}CACHE may even create "holes" in the data blocks (in PM and/or DRAM) due to partial overwrites that are not aligned with the block size. To ensure fast reads, P^{2}CACHE leverages *in-DRAM* indexes to facilitate data search, including indexes for (1) log entries in PM's metadata log; (2) data blocks in PM's file data log area; (3) data blocks in the page cache; and (4) partial-write slots in the page cache. These indexes enable the rapid assembly of read content, even when it is distributed across multiple storage media. Importantly, these indexes reside exclusively in DRAM and can be quickly reconstructed from the persistent cache, such as during a system reboot or recovery.

3.2 Write-centric Persistent Cache

To achieve high performance, it is crucial to defer "writes" to slow storage for as long as possible. This is precisely why most kernel file systems employ DRAM-based caches, such as dcache for metadata and the page cache for file data, to expedite I/O operations. As depicted in Figure 2, P^{2}CACHE introduces a new OS component, a PM-based *persistent cache*, positioned beneath the VFS and above any legacy kernel file systems. It interfaces with the VFS to efficiently persist updates to file system data and metadata, ensuring crash consistency and quickly responding to user applications. Meanwhile, P^{2}CACHE relies on mature, well-tested underlying kernel file systems for data organization and management – i.e., the persisted metadata/data update operations in the persistent cache are eventually flushed back to underlying file systems.

3.2.1 Layout of PM

Operation log: P^{2}CACHE's persistent cache captures and records file system operations related to writes (i.e., metadata/data updates) in a PM-backed write-ahead operation log (WAL). As illustrated in Figure 3, a WAL is implemented as a circular buffer consisting of fixed-size log entries. As there are two types of operations, directory operations and file operations, P^{2}CACHE builds two WALs: the directory write-ahead-log (dWAL) and the file write-ahead-log (fWAL). To improve the concurrency of P^{2}CACHE, each CPU core has its own WALs to log updates on that core independently.

	VFS Interface
Directory	create(), link(), unlink(), symlink(), mkdir(), rmdir(), mknod(), update_time(), setattr(), rename()
File	write(), write_iter(), fsync(), flush() open() (access time might need to updated)

Table 1: Interface exposed by VFS for metadata/data updates.

Algorithm 1 Atomically persisting file system updates

```
1:  function PERSISTENCE(operation)
2:      log_entry := create_a_log_entry(operation);
3:      if is_directory_op then
4:          write_to_dWAL(log_entry);
5:      else
6:          write_to_fWAL(log_entry);
7:          data_persistence(log_entry);   //Section 3.2.3
8:      end if
9:      update_in_DRAM_indexes();      //Section 3.3.2
10:     sfence();
11:     update_log_tail();
12:     sfence();
13: end function
```

PM space management: P^{2}CACHE partitions the PM space into n groups, where n is the number of CPU cores. Each group is further divided into two main areas: the *metadata log area* for storing dWALs and fWALs and the *file data log area* for storing data updates, as depicted in Figure 3. In systems with multiple PM DIMMs, P^{2}CACHE employs the *interleaved* mode, which distributes contiguous data blocks across the DIMMs in an interleaved manner. The management of PM space allows P^{2}CACHE to achieve high concurrency: (1) P^{2}CACHE-related tasks can be independently handled on different cores; (2) Sequential reads/writes can be concurrently served by multiple PM DIMMs. While file and directory operations can be processed by different cores and stored across multiple WALs, these operations are inherently ordered by their time of occurrence. Each operation log entry in the WAL contains a timestamp, as depicted in Figure 3. This guarantees that logged operations are later consumed by the underlying file system in the exact order in which they were issued by user applications. P^{2}CACHE relies on the VFS to prevent conflicts arising from concurrent updates to the same directory or file (via the per-inode read-write lock). For example, while one thread is writing data to a file, all other threads attempting to read from or write to the same file must wait. As a result, writes to the same file are recorded sequentially across the WALs, and their orders are determined by their timestamps.

3.2.2 Durability and Crash Consistency

Instant metadata/data durability: Using the PM-backed operation log, P^{2}CACHE first ensures instant *data/metadata durability*. Any metadata/data updates are captured by the persistent cache and *synchronously* persisted in the WALs. P^{2}CACHE captures these updates via the VFS-exposed interface as listed in Table 1. For each update, one or more log entries are synchronously created to store such an update operation in either the dWAL or the fWAL for metadata durability. If the operation involves the file data update (e.g., `write()`), new data should also be synchronously stored for data durability (more details in Section 3.2.3). Note that achieving instant metadata/data durability in traditional kernel file systems requires user applications to explicitly invoke synchronous operations, such as `fsync()` or `fdatasync()`. For example, most database systems use `fsync()` to ensure immediate data durability. In contrast, with P^{2}CACHE's instant data durability, (1) those `fsync()` issued by legacy applications can be immediately returned; (2) P^{2}CACHE-aware applications can eliminate `fsync()`; the return of a file operation indicates that both metadata and data have been persisted.

Strong crash consistency: P^{2}CACHE further provides *strong crash consistency* similar to PM-specialized file systems [43]. P^{2}CACHE achieves this by ensuring that each file operation is *atomic* – i.e., updates made by the operation are committed in an all-or-none fashion. As described in Algorithm 1, for a *metadata* update, P^{2}CACHE appends the operation to the end of the dWAL/fWAL by creating a log entry (size of 64 bytes). Then, P^{2}CACHE atomically updates the log tail to commit the metadata update. For a *data* update, P^{2}CACHE first appends the operation in the fWAL. Depending on the type of the writes (partial or full-block), P^{2}CACHE stores the file data either to the fWAL by creating log entries or in the file data log area by allocating free data blocks (Section 3.2.3). Finally, the log tail will be updated to commit the data update. Note that as Optane PM only guarantees atomicity for an 8-byte update, ensuring the atomicity of updates larger than 8 bytes (e.g., metadata/data updates) requires P^{2}CACHE to atomically move the log tail to the end of the dWAL/fWAL, thus committing the update. To ensure correct write ordering and prevent the tail update from occurring before the metadata/data update, P^{2}CACHE uses two `sfence` instructions: one after WALs or file data are written, and one after the log tail is updated.

Compared to in-kernel file system journaling (e.g., JBD2), which offers "relaxed" consistency, P^{2}CACHE's strong consistency provides the following benefits: (1) Each file operation mostly involves updating a small log entry (e.g., 64 bytes), which is much lightweight, whereas JBD2 needs a complex transaction operation involving multiple (4 KB) blocks, such as a journal header, multiple descriptor blocks, and a journal commit block. (2) Since the logs in P^{2}CACHE are committed to PM, they are persistent. As PM has a much larger capacity (than DRAM), persisted data can stay for quite a long time. Hence, P^{2}CACHE can defer "writes" to the underlying slow file systems and storage devices as long as possible. (3) Once operations are asynchronously consumed by the underlying file system, many optimizations can be employed, such as coalescing repeated writes and removing obsolete data, similar to [28]. (4) As shown in Section 4, P^{2}CACHE (though with strong consistency) achieves much higher performance than legacy kernel file systems (with relaxed consistency) because P^{2}CACHE significantly mitigates software overhead.

While P^{2}CACHE can implement "relaxed" consistency by placing WALs in DRAM first and asynchronously flushing

them back to PM, its current focus lies in providing strong (synchronous) consistency to kernel file systems. Further, P^{2}CACHE does not support atomic `mmap`, as "mmapped" I/Os bypass the VFS and access "mmapped" files directly via `load`/`store` instructions. Currently, P^{2}CACHE relies on user applications to achieve instant data durability and strong consistency for `mmap`, while a modified (or new) `mmap` interface for update atomicity is our ongoing investigation.

3.2.3 Fine-grained, Highly-efficient Data Logging

Compared to metadata, persisting data operations can incur much higher overhead. To mitigate such overhead, P^{2}CACHE invents a fine-grained, highly-efficient data logging mechanism that leverages PM's byte-addressability.

To ensure consistency, one should never overwrite old data before committing its new update. Otherwise, if a crash happens in the middle of an overwrite, it may corrupt the old data, causing inconsistency. To consistently persist a file data update, an intuitive approach is to allocate *free* data blocks in PM for storing the new file data, record the addresses of these blocks, and finally commit the data update. Up to this point, the data blocks that store old data can be released. Reclaiming these blocks can be done asynchronously (Section 3.4).

Note that if the updated file data *aligns* perfectly with one or multiple block boundaries (e.g., 4 KB), no data copying is required. In this case, the new file data blocks simply replace the old data blocks once the update is committed. If the updated file data does not align with the data blocks, *partial updates* are involved. Existing approaches [19, 23, 43] use a CoW strategy to copy the old data to a new data block and then apply the partial updates. Unfortunately, this approach leads to write amplification and long write latency.

P^{2}CACHE addresses this issue by decoupling (and delaying) "copy" from "write" in a CoW operation, named *decoupled CoW*. Decoupled CoW distinguishes writes of different sizes. As depicted in Figure 3, there are two types of partial writes: (1) For a *small* partial write (< 2KB, assuming a block size of 4 KB), P^{2}CACHE first appends the write operation log entry to the end of the fWAL and then directly appends the data content after the log entry. Finally, P^{2}CACHE atomically updates the log tail to the end of the data content to commit the update. (2) For a *large* partial write ($\geq$ 2KB and < 4KB), P^{2}CACHE instead allocates a free block to store the content of the partial write. Similar to NOVA [43], P^{2}CACHE employs a red-black tree for tracking and allocating free blocks.

In both the aforementioned cases, P^{2}CACHE does *not* copy the old data in the write path, neither does it in the read path – P^{2}CACHE devises an approach to efficiently assemble distinct partial updates during reads (Section 3.3.2). P^{2}CACHE performs data copying to convert a partially updated block to a full block at any later time. For instance, when reclaiming space in PM (Section 3.4), P^{2}CACHE copies small partial writes from the fWAL to their data blocks in the file data log area, or fills the missing portion in the data block of the large partial write with old data. If such data blocks do not exist in PM, a read-modify-write (RMW) operation is invoked to copy the required data from the underlying file system.

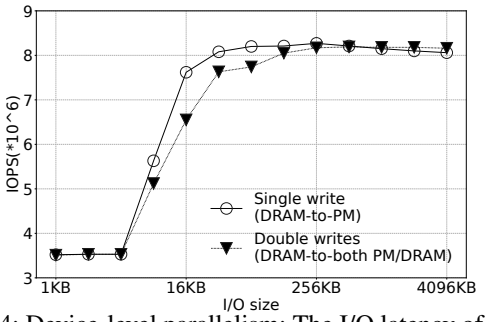


Figure 4: Device-level parallelism: The I/O latency of writing data to both PM and DRAM (i.e., one sfence after two writes: one to PM, and one to DRAM) is almost the same as that of writing data to PM only (i.e., one sfence after each PM write).

The decoupled CoW approach allows P^{2}CACHE to quickly persist partial updates by leveraging PM's byte-addressability. Real-world I/O traces show that a significant number of partial updates, ranging from 30% to 90% [33], commonly exist. In addition, by distinguishing writes by their sizes, P^{2}CACHE ensures that it requires copying *at most* half a block of data (e.g., 2KB with a block size of 4KB): P^{2}CACHE either copies the data of small partial writes (< 2KB) to their data blocks or the unmodified portion of old data (< 2KB) to the data blocks of large partial writes (> 2KB). Section 4 demonstrates that P^{2}CACHE, with the fine-grained data logging mechanism, achieves much higher performance for small writes than NOVA [43], a leading PM-specialized file system.

3.3 Read-centric Page Cache

P^{2}CACHE advances the page cache to handle most read I/Os in the tiered PM/DRAM hierarchy. P^{2}CACHE's page cache is a separate implementation other than the native page cache. It does not impact the behaviors of non-P^{2}CACHE supported kernel file systems, which still access the native page cache. To allow the persistent cache and P^{2}CACHE's read-centric page cache to work efficiently without comprising strong consistency, P^{2}CACHE employs a simple-and-effective *inclusive cache model* to exploit *device-level* parallelism.

3.3.1 Inclusive Cache Model

Similar to traditional caching mechanisms, P^{2}CACHE strives to maximize the hit ratio of the DRAM-based page cache. P^{2}CACHE employs an *inclusive cache model* where multiple copies of the same data can be stored across the tiered memory, and the topmost layer (i.e., DRAM) always contains the latest data version. It works as follows: (1) Given a *write* access, it will be persisted by the persistent cache (Section 3.2.2); meanwhile, a data copy will be made to the page cache before committing the update in PM. In this way, P^{2}CACHE allows the page cache to always have the latest version of all cached data. (2) Given a read access, P^{2}CACHE searches from the page cache (in DRAM), then the persistent cache (in PM), and finally the underlying file system until

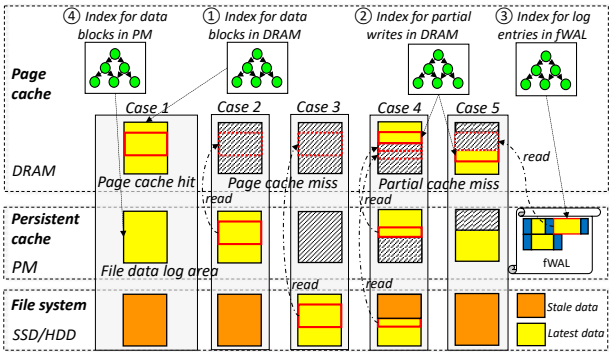


Figure 5: Fast read assembling via in-DRAM indexes.

it finds the data. For a page cache miss (e.g., the first time read or due to page eviction), the required data must be transferred either from the persistent cache or the underlying file system to the page cache, but data are never moved from the underlying file system to the persistent cache. (3) P^{2}CACHE uses LRU for cached page replacement, though other policies also apply. Evicted (dirtied) pages are dropped because any modifications have been recorded in the persistent cache.

Although P^{2}CACHE involves "double writes" for each data update, these two writes can be performed *in parallel* as they target separate devices: one for PM and one for DRAM. P^{2}CACHE can benefit from *device-level parallelism*. Figure 4 shows that the bandwidth (and latency) under "double writes" (i.e., writing to DRAM and PM at the same time) is very close to the "single-write" case (i.e., writing to PM only), indicating that the slower PM hides the latency of the extra data copy to DRAM as long as these two writes overlap each other. This way, P^{2}CACHE trades DRAM's bandwidth for simple synchronization between the two caches leading to a simplified read path. Note that DRAM has a much higher write bandwidth than PM (e.g., 6x [44]). Similar to the native page cache, P^{2}CACHE's page cache only uses "idle" DRAM and can grow/shrink. As all data on DRAM are in sync with PM, when the system memory pressure is high, the DRAM used by P^{2}CACHE can be reclaimed for other applications.

3.3.2 Fast Reads

While P^{2}CACHE's data logging (Section 3.2.3) greatly sharpens the write path, it brings new challenges to the read path due to: (1) While an operation log is efficient for updates, but not anymore for searching data (in the event of a page cache miss); (2) Decoupled CoW could leave holes in a page cache's data block – i.e., regions that are neither written by applications nor fetched from PM or underlying file systems.

In-DRAM indexes: To address these challenges, P^{2}CACHE uses in-DRAM indexes. P^{2}CACHE leverages Linux kernel's `XArray` – a memory-efficient, parallelizable treed data structure that performs lookups without locking – to create four per-`inode`, in-DRAM indexes (Figure 5) to track ① data blocks in the page cache; ② partial-write slots in the page cache; ③ log entries in the fWAL; and ④ data blocks in PM's file data log area. In consequence, in the *write* path, before committing a data update, P^{2}CACHE needs to insert the mapping information – between the updated data range and the log entry in the fWAL – in index ③. If a data block in the file data log area or the page cache has been allocated, index ④ or ① should be updated. If such an update involves a partial write, the offset and length of the partial write should be stored in index ②.

Assembling data for reads: With these in-DRAM indexes, the data content of a read request, specified by `offset` and `length`, can be quickly assembled as follows:

First, P^{2}CACHE uses the `offset` to query its per-inode index ① to check whether the data has been *fully* cached in the data block(s) of the page cache. If so, P^{2}CACHE returns the data of the requested length to user applications directly (e.g., case 1 in Figure 5). Otherwise, P^{2}CACHE uses the 2-tuple key `{offset, length}` to query index ② to check whether one or more partial slots in the range of the requested data exist in the page cache (e.g., case 4 & 5). If the aggregated partial slots do not cover the whole requested data, P^{2}CACHE moves to the persistent cache for the missing slots (e.g., case 4 & 5).

P^{2}CACHE uses the same 2-tuple key `{offset, length}` to query index ③ to get all log entries belonging to the queried data range, some of which may contain small partial writes (e.g., case 5). Further, by providing the `offset`, P^{2}CACHE retrieves the data block(s) stored in the file data log area via index ④ (e.g., case 2 & 4). Then, P^{2}CACHE copies the needed (missing) data slots – from the combined small partial writes (from the fWAL) and large partial writes (from PM's data blocks) – to the page cache (e.g., case 2, 4 & 5). If, unfortunately, there are still uncovered "holes" (e.g., case 3 & 4), P^{2}CACHE contacts the underlying file system for reading the needed data blocks to the page cache, taking longer time.

3.4 System Recovery and Digest

Rebuilding cache: P^{2}CACHE updates the log tail to commit operation-related records (Algorithm 1), indicating that all records preceding the log tail are considered valid. In case of a system crash or system remount, P^{2}CACHE discards any uncommitted records in the operation log (WALs). During system recovery/remount, *to facilitate fast reads*, P^{2}CACHE needs to scan the logs and build (1) two in-DRAM indexes, i.e., index ③ and index ④ (Section 3.3.2) and (2) a hash table (i.e., `dCache`). The process of scanning and building is considerably quick because logs are typically small (no data scanning is needed) and stored in fast PM. Table 2 shows that given a practical setup – for instance, with 10 thousand opened directories/files and 1 million log entries (typically multiple updates target one directory/file) – P^{2}CACHE uses a single core to recover index ③ within 33 ms while less than 17 ms to rebuild `dCache`. Moreover, the recovery of in-DRAM indexes can be made in parallel due to the design of per-core WALs – the time to rebuild ③ drops to 14 ms with 8 cores.

Digesting cache: P^{2}CACHE applies cached operations in PM to underlying file systems *asynchronously* via the existing I/O interface, namely the digest process [28]. The large capacity

# of log entries/files	10^4	10^5	10^6	10^7
Rebuild dCache (ms)	16.52	95.57	904.84	8009.02
Rebuild index ③ (ms)	0.89	4.44	32.92	320.50

Table 2: Time to rebuild dCache and in-DRAM indexes.

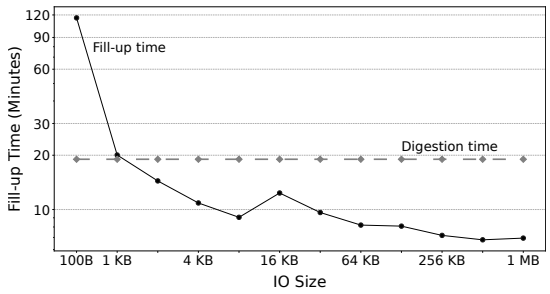


Figure 6: P^{2}CACHE filling-up time: P^{2}CACHE operates on a 512-GB PM, while the kernel file system (Ext4) runs upon one 2-TB SSD. A (sequential) write-intensive workload from Filebench [6] with 8 threads continuously issues I/O requests of various sizes at *maximum* speed. The PM's speed determines the cache fill-up time (the solid line), while the SSD's speed determines the digestion time (the dashed line). The digest process can merge adjacent write requests in the persistent cache and create large sequential I/Os writing to SSD.

of PM provides more flexibility to P^{2}CACHE by preventing the contention between the digest process and the normal working process. P^{2}CACHE digests operations during system idleness upon two conditions: the usage of PM space is high (e.g., more than 80%) or the number of log entries is large (e.g., more than 10 million). During the digest process, P^{2}CACHE relies on the return of `fsync` (after each operation) to ensure that metadata/data has been persisted and committed. A system crash may also occur during digest; P^{2}CACHE simply re-applies uncommitted operations. Similar to [28], P^{2}CACHE applies optimizations to coalesce multiple updates to the same file/directory during the digest process.

Filling cache: Given that (1) PM is generally faster than the storage devices on which the underlying kernel file systems operate (e.g., HDD/SSD) and (2) the speed of the digest process is mainly limited by the kernel file system's storage devices, the persistent cache of P^{2}CACHE can reach its full capacity. As a concrete example, Figure 6 shows the time required to fill up the persistent cache. When the persistent cache is full, P^{2}CACHE's current strategy is to throttle foreground I/O threads till the digest process reclaims enough space from PM (e.g., 20%). We note that as long as the *average* I/O rate is lower than the speed of the kernel file system's storage devices – or the fill-up time (the solid line in Figure 6) exceeds the digestion time (the dashed line) – the digest process can write back data timely. In practice, the occurrence of P^{2}CACHE reaching PM's full capacity is expected to be less frequent because (1) PM could be large (further CXL can aggregate even larger PM), and (2) I/O requests in a production environment arrive at moderate rates [38], typically lower than the speed of the kernel file system's storage devices.

4 Evaluation

We have implemented P^{2}CACHE as a Linux kernel module with ∼2000 lines of kernel code. P^{2}CACHE is available at https://github.com/YesZhen/P2CACHE. As an independent kernel module, P^{2}CACHE can be easily (un-)loaded without modifying other kernel components. P^{2}CACHE has passed all the test cases (∼7,000) of Linux official POSIX file system test suite [13], demonstrating P^{2}CACHE's POSIX compliance.

We have evaluated the effectiveness of P^{2}CACHE. Results from microbenchmarks demonstrate that (1) P^{2}CACHE accelerates metadata operations by *∼200x* against kernel file systems (e.g., Ext4 with `fsync`) and *3.5x* against PM-specialized file systems (e.g., NOVA). (2) P^{2}CACHE yields much higher write performance, particularly for *small, partial* writes – e.g., by *6.8x* than NOVA and *1,000x* than Ext4 (with `fdatasync`) for 1 KB writes. (3) P^{2}CACHE can leverage the DRAM-based page cache to achieve higher read performance – by *1.5x* than NOVA. The performance benefits brought by P^{2}CACHE further contribute to the improved application-level performance – e.g., by *72%* to NOVA for RocksDB's `insert` operations.

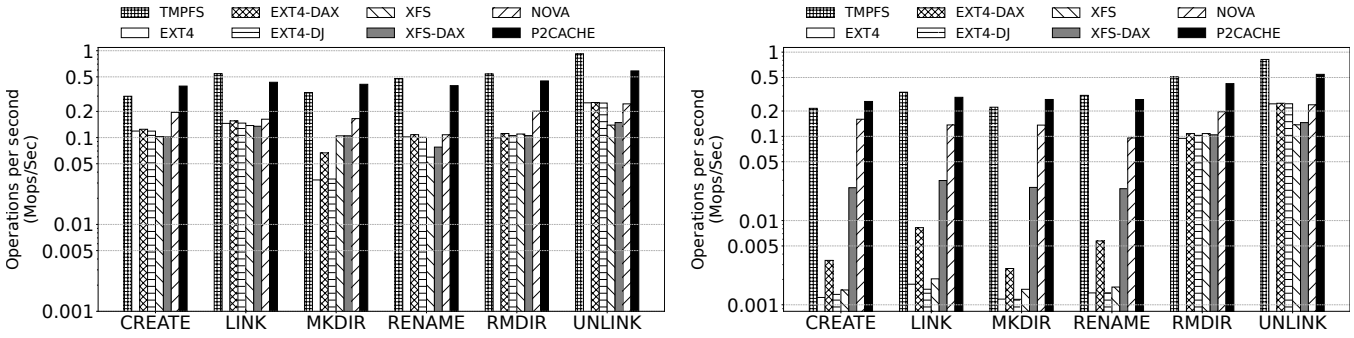
Experimental setup: The experiments were conducted on an ASUS RS700-E10-RS12U server equipped, with two 12-core Intel Xeon Gold 5317 processors (3.0 GHz and 18M Cache) with 2 NUMA nodes each with 256 GB DRAM. Hyperthreading was disabled while turbo boost was enabled. We installed four 128 GB (totaling 512 GB) Intel Optane 200 series persistent memory for each NUMA node and one 2-TB Samsung PM883 SSD. Since our focus is not on the NUMA effect, all experiments were conducted on one NUMA node.

We evaluated P^{2}CACHE on the Linux kernel 5.4, comparing it with (1) two kernel file systems, Ext4 and XFS, both operating on the SSD with the default metadata journaling mode and the data journaling mode for Ext4 (i.e., Ext4-DJ); (2) two PM-enhanced file systems, Ext4-DAX [5] and XFS-DAX [25], operating on PM; and (3) one PM-specialized file system, NOVA [43] (in strict mode) also operating on PM. We tested P^{2}CACHE atop Ext4 as the underlying kernel file system, though P^{2}CACHE can run atop any kernel file system. We evaluated P^{2}CACHE with both microbenchmarks and real-world applications. We have developed our own microbenchmarks to delicately generate desired I/O requests and patterns to test various design aspects of P^{2}CACHE. We selected three representative real-world applications for testing: Filebench [6], RocksDB [1], and MinIO [14].

4.1 Microbenchmarks

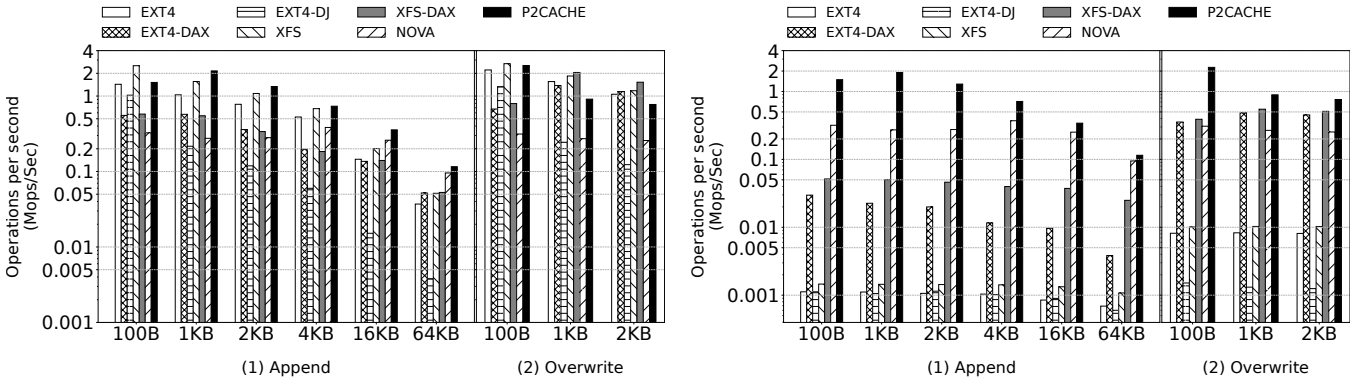
Metadata operations: We first show how P^{2}CACHE benefits metadata operations. We chose the six most complex ones in Figure 1, i.e., `create`, `link`, `mkdir`, `rename`, `rmdir`, and `unlink`. For each type of metadata, our micro-benchmark kept issuing the operations sequentially – i.e., the subsequent one was issued upon the completion of the previous one.

Figure 7a shows that P^{2}CACHE significantly accelerates the speed of all six metadata operations compared to all other cases, except for `tmpfs`. For example, for the most complex



(a) Without `fsync`. (b) With `fsync`.

Figure 7: P^{2}CACHE significantly accelerates metadata operations as against other cases except for `TMPFS`.



(a) Without `fdatasync`. (b) With `fdatasync`.

Figure 8: P^{2}CACHE accelerates data operations, especially for small, partial writes, as against other cases.

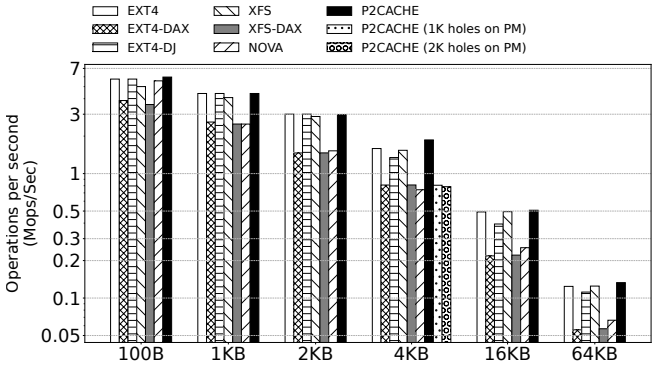


Figure 9: Comparisons of performance for reads.

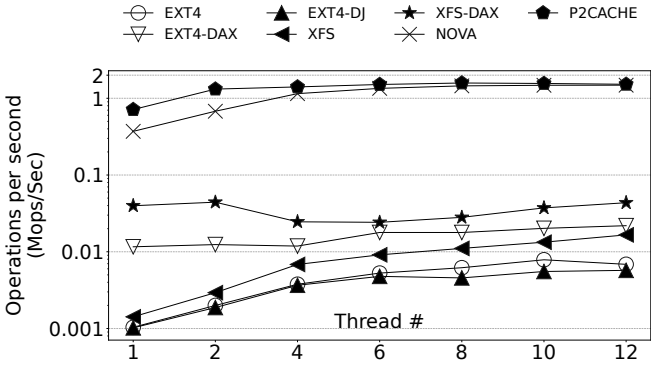


Figure 10: Scalability test with 4 KB append operations.

operation `rename` (i.e., involving multiple inodes), P^{2}CACHE achieves *∼4x* the performance of NOVA and Ext4 (in terms of operations/second). It is because P^{2}CACHE keeps the critical path (involving PM) extremely short by simply storing a log entry that represents the `rename` operation in the dWAL. In contrast, NOVA requires the creation of multiple logs and updates to multiple log entries, while Ext4 involves more operations (i.e., first `unlink` and then `link`). As another example, P^{2}CACHE enhances the performance of `mkdir` by a factor of *12x* and *6x* compared to Ext4 and Ext4-DAX. Figure 7a also shows that the performance gap between P^{2}CACHE and `tmpfs` is narrow – `tmpfs` is an extremely simple kernel file system that works upon DRAM. P^{2}CACHE is mostly within *80%* the performance of `tmpfs`; P^{2}CACHE even outperforms `tmpfs` for `create` (1.3x) and `mkdir` (1.25x). The main reason lies in that, instead of using the default heavyweight "inode_init_always" function, P^{2}CACHE implements its optimized one by initializing (fewer) needed fields. Again, as a caching mechanism, P^{2}CACHE can be lighter than full-fledged file systems, even including `tmpfs`.

Figure 7b shows a *strong consistency* scenario by issuing `fsync` after each metadata operation. Note that P^{2}CACHE and NOVA provide strong consistency in nature; they return `fsync` without any action. With strong consistency, ex-

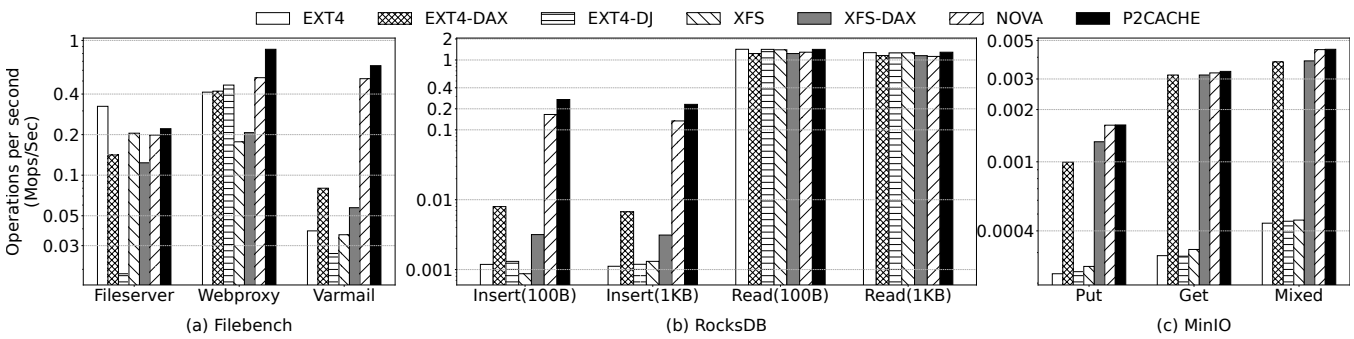


Figure 11: Performance comparisons of using real-world applications (a) Filebench, (b) RocksDB, and (c) MinIO.

cept for P^{2}CACHE, NOVA, and `tmpfs`, the performance of all other approaches drops significantly, where P^{2}CACHE outperforms them by up to *200x*. Noticeably, the performance of P^{2}CACHE and NOVA drops with `fsync` (Figure 7b) compared to the case without `fsync` (Figure 7a). It is because the `fsync` system call (though a no-op) incurs higher software overhead. As all the approaches rely on VFS's `dCache` for caching metadata, they achieve the same performance for read-related metadata operations (not listed).

Data operations: Next, we demonstrate how P^{2}CACHE benefits data operations. Figure 8a shows that in most cases P^{2}CACHE significantly outperforms other PM-based approaches in write performance – under both `append` (i.e., sequentially writing to the end of a file) and `overwrite` (i.e., sequentially overwriting existing content) – due to its lightweight design for the write path. Particularly, P^{2}CACHE achieves high performance for partial writes (e.g., for 100 bytes, 1 KB, and 2 KB) due to its *fined-grained, highly-efficient* data logging mechanism (Section 3.2.3). For example, the performance of `append` (for 1 KB) under P^{2}CACHE is 7.8x, 3.7x, and 3.9x as high as NOVA, Ext4-DAX, and XFS-DAX. Note that, only P^{2}CACHE and NOVA provide strong consistency, while Ext4-DAX and XFS-DAX only provide metadata consistency. Even with strong consistency, in many cases (except for the 100B case for `append` and 1KB & 2KB cases for `overwrite`), P^{2}CACHE achieves higher performance than kernel file systems (Ext4 and XFS), where the `append` and `overwrite` operations are directly applied to DRAM-based page cache. It indicates that P^{2}CACHE greatly reduces software overhead. The reason that the 100 B `append` case does not perform as well as other partial-write cases (e.g., 1 KB and 2 KB) lies in that the I/O size of 100 bytes is not aligned with PM's physical media access granularity, i.e., 256 bytes, causing write amplification and inefficiency [44].

Similarly, Figure 8b shows a strong consistency scenario by issuing `fdatasync` after each `append` or `overwrite` operation. The performance gap between P^{2}CACHE and others (except NOVA) widens significantly – e.g., P^{2}CACHE outperforms Ext4/XFS by more than *1,000x* and Ext4-DAX/XFS-DAX by more than *10x* for small writes. The poor performance of the kernel file systems is due to (1) slow SSD and

Metadata/data ops	Initial state	Steady state	Steady/initial
CREATE	259,697	300,778	+15.8%
LINK	290,852	276,509	-4.9%
MKDIR	274,165	308,090	+12.4%
RENAME	273,929	250,650	-8.5%
RMDIR	422,855	376,619	-10.9%
UNLINK	544,964	513,720	-5.7%
Append (100 B)	1,417,632	1,428,365	+0.8%
Append (64 KB)	114,357	128,625	+12.5%
Overwrite (100 B)	2,266,327	2,297,884	+1.4%
Overwrite (64 KB)	118,416	132,006	+11.5%

Table 3: Performance comparisons between the initial and steady states: metadata (Mops); data (IOPS).

(2) high overhead of file system journaling (Section 2.2).

For P^{2}CACHE's read performance, we tested Case 1 and 5, as listed in Figure 5. For Case 1, where all data was cached in the page cache, we measured the performance of sequential reads with various I/O sizes ranging from 100 B to 64 KB. For Case 5, where partial data was stored in DRAM and partial data was in PM, we randomly created numerous 1 KB or 2 KB "holes" in the data blocks of the page cache (averaging one hole per 4 KB) and measured the performance of sequential reads with the I/O size of 4 KB.

Figure 9 shows that, for case 1, P^{2}CACHE achieves the same (or slightly better) performance as (than) those which can leverage the DRAM-based page cache, e.g., Ext4, Ext4-DJ, and XFS. However, the difference is that P^{2}CACHE also provides strong consistency, while others do not. In contrast, P^{2}CACHE outperforms other PM-related approaches that bypass the page cache, e.g., by 1.5x compared to NOVA (for 4 KB reads). For case 5, P^{2}CACHE quickly assembled each 4 KB read from both DRAM and PM (Section 3.3.2). However, the read performance was limited by the slower device – e.g., PM. For example, P^{2}CACHE achieves the same read performance as NOVA, Ext-DAX, and XFS-DAX – the PM's speed limited the read performance.

Steady-state performance: We studied the steady-state performance by first simulating a steady-state scenario of P^{2}CACHE through running a mix of metadata/data operations until the persistent cache gradually reaches its 60% capacity with around 1 million files. We then measured the performance by running the above six metadata operations and append/overwrite data operations with strong consistency

(`fsync` or `fdatasync` after each operation) and compared the steady-state performance with that of the "initial state", where the persistent cache is empty. Table 3 demonstrates that the performance variation mostly falls within 15%.

Concurrency and scalability: We measured the scalability of P^{2}CACHE by running an increasing number of concurrent threads accessing different files on separate cores (up to 12 as one NUMA node has 12 cores). The results in Figure 10 show that both P^{2}CACHE and NOVA scale well as the number of threads increases until they reach the peak PM performance (for 4 KB `appends`). P^{2}CACHE achieves peak performance much faster than NOVA due to, again, its lightweight design.

Consistency checks: We developed a consistency checker to empirically generate test cases to examine whether the strong consistency property provided by P^{2}CACHE holds. The checker added "crash points" along P^{2}CACHE's persistence path (Algorithm 1) with three cases (1) inserting the crash point before the first `sfence`; (2) between the two `sfences`; and (3) after the second `sfence`. For Case 1, the operation should not be atomically persisted as the log tail is not updated; for Case 3, the operation should be persisted as the log tail is updated; for Case 2, the operation may or may not be persisted. The checker examines that if the operation is persisted, the final state should match the expected state (a priori knowledge), while if the operation is not persisted, nothing should be recorded (i.e., none or nothing). For all the microbenchmark tests, we also used the checker to perform consistency checks without observing any violations. We note that such consistency checks are incomplete and unable to explore all possible test cases. We leave the investigation of a more comprehensive method in future work.

4.2 Real-world Applications

Filebench: To evaluate the performance of P^{2}CACHE with real-world applications, we first selected three Filebench workloads [6]: (1) a write-intensive workload, *fileserver* (1:2 read/write ratio); (2) a read-intensive workload, *webproxy* (5:1 read/write ratio); and (3) a read/write balanced workload, *varmail* (1:1 read/write ratio). For all cases, the average read size was 1 MB, and the average write size was 16 KB. We added another type of write with the I/O size of 1 MB for fileserver. We fixed the thread number to 8 for all cases.

Figure 11 (a) shows that P^{2}CACHE consistently outperforms other PM-based approaches (e.g., NOVA, Ext4-DAX, and XFS-DAX), especially for read-intensive test cases, e.g., webproxy and varmail, due to P^{2}CACHE's read/write distinguishable memory hierarchy which leverages both PM and DRAM. For example, P^{2}CACHE outperforms NOVA by 60% for webproxy and 20% for varmail. P^{2}CACHE achieves lower performance than Ext4 for fileserver with intensive write operations, as Ext4 leverages faster DRAM-based page cache while P^{2}CACHE persists data in slower PM for writes. However, Ext4 does not provide strong consistency. P^{2}CACHE outperforms Ext4-DJ (with data journaling enabled) by *10x*.

RocksDB: We then used *db_bench* [1] – RocksDB's official benchmark tool – to evaluate P^{2}CACHE for RocksDB (a key-value store). RocksDB's architecture is highly concurrent for reads but not for writes [8]. Therefore, for writes, we focused on a single-threaded *synchronous* case by randomly inserting 10 million records to RocksDB; for reads, we focused on a multi-threaded random case with 8 threads to randomly read 10 million key-value records from RocksDB. We prepared a dataset with 10 million records. We fixed the key size to 20 bytes and evaluated two value sizes – 100 B and 1KB – a common case in RocksDB.

Figure 11 (b) shows that P^{2}CACHE outperforms all other approaches for small writes (i.e., `insert`) – e.g., by ∼72% to NOVA, ∼33x to Ext4-DAX, and ∼200x to Ext4. Note that, the extremely poor performance of Ext4 (though using the native page cache) is due to (1) synchronous insert operations, which persist data on slow SSD; and (2) read-modify-write caused by unaligned writes (e.g., 100 B or 1 KB are not aligned with 4 KB block size). As we purposely conducted the tests of reads after `insert` to have all records stored in the page cache, all the approaches (e.g., P^{2}CACHE, Ext4, Ext4-DJ, and XFS) that can leverage the page cache achieve the same performance – higher than PM-based approaches that bypass DRAM.

MinIO: Last, we evaluated P^{2}CACHE using an object storage, MinIO [14]. Compared to the above applications (e.g., RocksDB and Filebench), MinIO's software I/O path is much longer with extra data management (e.g., data checksum and placement). We used MinIO's official benchmark tool *warp* [16] with three workloads: `put`, `get`, and `mixed` (45% get, 30% stat, 15% put, and 10% deletion operations). We used log2 to distribute object sizes – i.e., objects are distributed in equal numbers for each doubling of the size.

Figure 11(c) shows that P^{2}CACHE and NOVA achieve the same performance under all cases and outperform other approaches by a range between 5% (over Ext4-DAX/XFS-DAX for `get`) and 10x (over Ext4/Ext4-DJ for `get`). We observed that software overhead from MinIO became dominant; neither P^{2}CACHE nor NOVA can exploit the full capacity of PM.

5 Conclusions

We have presented P^{2}CACHE, an in-kernel caching mechanism, which harnesses performance benefits and unique characteristics of fast, byte-addressable PM for legacy kernel file systems. P^{2}CACHE works upon a read/write-distinguishable memory hierarchy that leverages PM to persist writes and DRAM to handle reads, thus equipping kernel file systems with the key properties similar to PM-specialized file systems, including instant data durability, strong consistency, high concurrency, and high performance. Our evaluation with both microbenchmarks and applications shows that P^{2}CACHE significantly increases the performance of legacy kernel file systems, and even higher than PM-specialized file systems.

6 Acknowledgments

We thank our shepherd Mike Mesnier and the anonymous reviewers for their helpful feedback. This work was supported by NSF under Awards CCF-1845706 and CNS-2237966.

References

[1] Benchmarking rocksdb. https://github.com/EighteenZi/rocksdb_wiki/blob/master/Benchmarking-tools.md.

[2] Build ultra high-performance storage applications with the storage performance development kit. https://spdk.io/.

[3] Ceph fs dynamic metadata management. https://docs.ceph.com/en/latest/cephfs/dynamic-metadata-management/.

[4] Compute express link™: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/download-the-specification.

[5] Direct access for files. "https://www.kernel.org/doc/html/latest/filesystems/dax.html".

[6] Filebench. https://github.com/filebench/filebench.

[7] How windows ntfs finally made it into linux. https://www.theregister.com/2021/10/13/how_ntfs_finally_made_it/.

[8] Improving rocksdb's write scalability counting things at smyte. https://www.heavybit.com/library/article/improving-rocksdbs-write-scalability-counting-things-at-smyte.

[9] Intel optane dc ssd series. https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds/optane-dc-ssd-series.html.

[10] Intel optane dimm. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[11] Intel optane is winding down. what's that mean for you your customers? https://www.techproviderzone.com/cloud-and-data-centers/intel-optane-is-winding-down-what-s-that-mean-for-you-your-customers.

[12] Intel® optane™ ssd p5800x series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html.

[13] Linux posix file system test suite. https://lwn.net/Articles/276617/.

[14] Multi-cloud object storage. https://min.io/.

[15] *The Linux Journalling API*. https://www.kernel.org/doc/html/latest/filesystems/journalling.html.

[16] Warp. https://github.com/minio/warp.

[17] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, Association for Computing Machinery.

[18] CHEN, Y., SHU, J., OU, J., AND LU, Y. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage 14*, 1 (apr 2018).

[19] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 133–146.

[20] CORBET, J. The multiqueue block layer. LWN.net.

[21] CORBET, J. Two new block i/o schedulers for 4.12. LWN.net.

[22] CORBET, J. The future of dax. LWN.net.

[23] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. EuroSys '14, Association for Computing Machinery.

[24] HELLWIG, C. Xfs for linux. In *Proceedings of Linux 2003 Conference and Tutorials, Edinburgh, Scotland* (2003).

[25] INTERNATIONAL., S. G. Xfs: A high-performance journaling filesystem. In *http://oss.sgi.com/projects/xfs*.

[26] JUNG, M. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). HotStorage '22, Association for Computing Machinery, p. 45–51.

[27] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 494–508.

[28] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 460–477.

[29] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 273–286.

[30] LEE, E., BAHN, H., AND NOH, S. H. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (USA, 2013), FAST'13, USENIX Association, p. 73–80.

[31] LI, S. block: An iops based ioscheduler. LWN.net.

[32] LIU, J., REBELLO, A., DAI, Y., YE, C., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 819–835.

[33] LU, H., SALTAFORMAGGIO, B., XU, C., BELLUR, U., AND XU, D. Bass: Improving i/o performance for cloud block storage via byte-addressable storage stack. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), pp. 169–181.

[34] MA, A., DRAGGA, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND MCKUSICK, M. K. Ffsck: The fast file-system checker. *ACM Trans. Storage 10*, 1 (jan 2014).

[35] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., AND VIVIER, L. The new ext 4 filesystem : current status and future plans.

[36] PARK, D., AND SHIN, D. iJournaling: Fine-Grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 787–798.

[37] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 1–32.

[38] SONG, H., KIM, S., KIM, J. H., PARK, E. J., AND NOH, S. H. First responder: Persistent memory simultaneously as high performance buffer cache and storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 839–853.

[39] VALENTE, P., AND ANDREOLINI, M. Improving application responsiveness with the bfq disk i/o scheduler. In *Proceedings of the 5th Annual International Systems and Storage Conference* (New York, NY, USA, 2012), SYSTOR '12, Association for Computing Machinery.

[40] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, Association for Computing Machinery.

[41] WU, K., GUO, Z., HU, G., TU, K., ALAGAPPAN, R., SEN, R., PARK, K., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (2021), pp. 307–323.

[42] WU, X., QIU, S., AND NARASIMHA REDDY, A. L. Scmfs: A file system for storage class memory and its extensions. *ACM Trans. Storage 9*, 3 (aug 2013).

[43] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 323–338.

[44] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[45] ZHONG, Y., LI, H., WU, Y. J., ZARKADAS, I., TAO, J., MESTERHAZY, E., MAKRIS, M., YANG, J., TAI, A., STUTSMAN, R., AND CIDON, A. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 375–393.

A Artifact Appendix

Abstract

The artifact contains the source code of P^{2}CACHE required to reproduce the results and figures presented in the paper. The code is designed to work upon Intel Optane PMem 200 series. To facilitate the reproduction of the results, we provide a collection of scripts for compiling and installing P^{2}CACHE, executing the experiments, collecting logs, and creating graphs. More details are available in the "README.md" file.

A.1 Description & Requirements

A.1.1 How to access

- **Link**: https://github.com/YesZhen/P2CACHE.git
- **Artifact license**: GNU GPL V3.0
- **Artifact version**: v0.0

A.1.2 Dependencies

For information on the hardware/software requirements needed to run P^{2}CACHE, please refer to "README.md".

A.1.3 Benchmarks

The experiments are carried out using several third-party benchmarking tools and applications, including FxMark, Filebench, db_bench/RocksDB, and warp/MinIO.

A.2 Testbed Setup

For instructions on how to set up and configure the test machine, please refer to the "README.md" file.

A.3 Evaluation

A.3.1 Major Claims

We summarize the major claims (Cx) in the paper as follows.

- (**C1**): P^{2}CACHE accelerates metadata operations, e.g., by *∼200x* against kernel file systems (e.g., Ext4) and *∼3.5x* against PM-specialized file systems (e.g., NOVA).
- (**C2**): P^{2}CACHE achieves much higher write performance especially for *small, partial* writes, e.g., by *6.8x* than NOVA and *1,000x* than Ext4 (with `fdatasync`) for 1 KB writes.
- (**C3**):P^{2}CACHE can leverage DRAM-based page cache to achieve high read performance, e.g., by *1.5x* than NOVA.
- (**C4**): The performance benefits brought by P^{2}CACHE further contribute to improved application-level performance – e.g., by *72%* to NOVA for RocksDB's `insert`.

A.3.2 Experiments

To reproduce the results presented in this paper, please refer to the "README.md" file and follow the instructions provided in the "Reproduce results from the paper" section.

Experiment (E1): Metadata Operations (without `fsync`)
Expected outcome. E1 produces the results as shown in Figure 7a, which illustrates that P^{2}CACHE significantly accelerates the speed of all six metadata operations (i.e., `create`, `link`, `mkdir`, `rename`, `rmdir`, and `unlink`) when compared to all other approaches (i.e., `Ext4`, `Ext4-DJ`, `XFS`, `Ext4-DAX`, `XFS-DAX`, and `NOVA`), except for `tmpfs`.

Experiment (E2): Metadata Operations (with `fsync`)
Expected outcome. E2 creates the results of Figure 7b. It demonstrates a strong consistency case by issuing `fsync` after each metadata operation. Except for P^{2}CACHE, `NOVA`, and `tmpfs`, the performance of all other approaches (i.e., `Ext4`, `Ext4-DJ`, `XFS`, `Ext4-DAX`, `XFS-DAX`) drops significantly.

Experiment (E3): Write Operations (no `fdatasync`)
Expected outcome. E3 generates the results as depicted in Figure 8a. It shows that in most cases (across various I/O sizes), P^{2}CACHE outperforms other PM-based approaches (i.e., `NOVA`, `Ext4-DAX`, and `XFS-DAX`) for two write operations: `append` and `overwrite`.

Experiment (E4): Write Operations (with `fdatasync`)
Expected outcome. E4 produces the results as depicted in Figure 8b. It demonstrates a strong consistency scenario by issuing `fdatasync` after each `append` or `overwrite` operation. P^{2}CACHE outperforms all other approaches in terms of higher `append` and `overwrite` performance.

Experiment (E5): Read Operations
Expected outcome. E5 creates the results as demonstrated in Figure 9. It shows that P^{2}CACHE achieves the same (or slightly better) performance as (than) those which can leverage the DRAM-based page cache (i.e., `Ext4`, `Ext4-DJ`, and

XFS). Further, P^{2}CACHE outperforms all other PM-based approaches (i.e., NOVA, Ext4-DAX, XFS-DAX) in terms of higher read performance.

Experiment (E6): Scalability
Expected outcome. E6 generates the results in Figure 10, showing that both P^{2}CACHE and NOVA scale well as the number of threads increases until reaching the peak performance, whereas P^{2}CACHE achieves the peak faster than NOVA.

Experiment (E7): Application: Filebench
Expected outcome. E7 produces the results as shown in Figure 11(a). It runs three workloads with Filebench, which are fileserver, webproxy, and varmail. The results show that P^{2}CACHE consistently outperforms other PM-based approaches (i.e., NOVA, Ext4-DAX, and XFS-DAX) in terms of higher application-level performance (operations/second).

Experiment (E8): Application: RocksDB
Expected outcome. E8 creates the results in Figure 11(b). It shows that for the insert operations (with sizes of 100 B and 1 KB), P^{2}CACHE outperforms all other approaches. All the approaches (i.e., P^{2}CACHE, Ext4, Ext4-DJ, and XFS) that can leverage page cache achieve similar read performance – slightly higher than other PM-based approaches that bypass DRAM (i.e., NOVA, Ext4-DAX, and XFS-DAX).

Experiment (E9): Application: MinIO
Expected outcome. E9 produces the results in Figure 11(c). The results show that P^{2}CACHE and NOVA achieve the same performance under all test cases and outperform other approaches (e.g., Ext4, Ext4-DJ, Ext4-DAX, XFS, and XFS-DAX).

Revisiting Secondary Indexing in LSM-based Storage Systems with Persistent Memory

Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu*

Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

Abstract

LSM-based storage systems are widely used for superior write performance on block devices. However, they currently fail to efficiently support secondary indexing, since a secondary index query operation usually needs to retrieve multiple small values, which scatter in multiple LSM components. In this work, we revisit secondary indexing in LSM-based storage systems with byte-addressable persistent memory (PM). Existing PM-based indexes are not directly competent for efficient secondary indexing. We propose PERSEID, an efficient PM-based secondary indexing mechanism for LSM-based storage systems, which takes into account both characteristics of PM and secondary indexing. PERSEID consists of (1) a specifically designed secondary index structure that achieves high-performance insertion and query, (2) a lightweight hybrid PM-DRAM and hash-based validation approach to filter out obsolete values with subtle overhead, and (3) two adapted optimizations on primary table searching issued from secondary indexes to accelerate non-index-only queries. Our evaluation shows that PERSEID outperforms existing PM-based indexes by 3-7× and achieves about two orders of magnitude performance of state-of-the-art LSM-based secondary indexing techniques even if on PM instead of disks.

1 Introduction

Log-Structured Merge trees (LSM-trees) feature outstanding write performance and thus have been widely adopted in modern key-value (KV) stores, such as RocksDB [22] and Cassandra [1]. Different from in-place update storage structures (e.g., B$^+$-Tree), LSM-trees buffer writes in memory and flush them to storage devices in batches periodically to avoid random writes, which enables high write performance and low device write amplification. Besides high write performance, many database applications also require high-performance queries on not only primary keys but also other specific values [11], thus necessitating secondary indexing techniques.

LSM-trees' attributes make it challenging to design efficient secondary indexing. Modern LSM-based storage systems typically store a secondary index as another LSM-tree [47] (e.g., a column family in RocksDB [44]). However, designed for block devices and optimized for write performance, LSM-trees are not competent data structures for secondary indexes which require high search performance. First, since secondary indexes usually only store primary keys instead of full records[1] as values, KV pairs in secondary indexes are small. LSM-trees' heavy lookup operations are inefficient for these small KV pairs. Second, secondary keys are not unique and can have multiple associated primary keys. LSM-trees' out-of-place write pattern will scatter these non-consecutive-arrived values (i.e., associated primary keys) to multiple pieces at different levels. Consequently, query operations need to search all levels in the LSM-based secondary index to fetch these value pieces. Besides the device I/O overhead, LSM-trees have non-negligible overheads of CPU and memory (i.e., indexing and Bloom filter) [17, 20, 34].

Moreover, the consistency of secondary indexes is another issue in LSM-based storage systems. As an LSM-based primary table adopts the blind-write pattern to update or delete records (appends new data without checking old data, versus read-modify-write in B$^+$-Trees) for high write performance, it is unable to delete the obsolete entry in a secondary index without acquiring the old secondary key. Consequently, when querying a secondary index, the system should validate each entry by checking the primary table before returning the results to users, which introduces many unnecessary but expensive lookups on the primary table for obsolete entries. Some systems fetch old records when updating or deleting records to keep secondary indexes up-to-date synchronously [9, 44], whereas this method discards the blind-write attribute and thus degrades the write performance.

Though many efforts have been made to optimize these predicaments [36, 40, 47, 50], they are difficult to solve the problems discussed above well, sacrificing either write per-

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

[1] For clarity, we use *record* to refer to a KV pair in the primary table, and *entry* to refer to a KV pair in a secondary index.

formance of the LSM-based storage systems or query performance of the secondary index.

As secondary indexing demands low-latency queries and the KV pairs of secondary indexes are small, we argue that leveraging persistent memory (PM) to provide a new solution for secondary indexing is promising. PM has many attractive advantages such as byte-addressability, DRAM-comparable access latency, and the ability of data persistency, which is well suited to secondary indexing. Though there are many state-of-the-art PM-based indexes [13, 25, 31, 33, 37, 38, 45, 46, 61, 62], none of them are designed for secondary indexing. Without considering the non-unique feature of secondary indexes and consistency in LSM-based KV stores, simply adopting existing general PM-based indexes as secondary indexes can overshadow their performance.

In this work, we propose PERSEID, a new persistent memory-based secondary indexing mechanism for LSM-based KV stores. PERSEID contains `PS-Tree`, a specifically designed data structure on PM for secondary indexes. `PS-Tree` can leverage state-of-the-art PM-based indexes and enhance them with a specific value layer, which considers the characteristics of both PM and secondary indexing. The value layer of `PS-Tree` works in a manner of blended log-structured approach and B$^+$-Tree leaf nodes, which is both PM-friendly and secondary-index-friendly. Specifically, new values are appended to value pages for efficient insertion on PM. During the value page split, multiple values (i.e., associated primary keys) that belong to the same secondary key are reorganized to store continuously for efficient querying.

Moreover, PERSEID retains the blind-write attribute of LSM-based KV stores for high write performance without sacrificing secondary index query performance. This is achieved by a lightweight hybrid PM-DRAM and hash-based validation approach in PERSEID. PERSEID uses a hash table on PM to record the latest version of primary keys. However, multiple random accesses on PM still incur high latencies. Thus, PERSEID adopts a small mirror of the validation hash table on DRAM which only contains useful information for validation. During validation, the volatile hash table absorbs random accesses to PM, and thus reduces the validation overhead. The small volatile hash table not only saves DRAM memory space but also reduces cache pollution.

PERSEID has a fairly low latency of index-only query[2] However, the overhead of non-index-only queries is still dominated by the LSM-based primary table. Therefore, we further propose two optimizations for non-index-only queries in PERSEID. First, as querying the primary table issued by the secondary index is an internal operation, we can locate KV pairs with additional auxiliary information much more efficiently, reducing cumbersome indexing operations. By matching the tiering compaction strategy [19, 41], we can further bypass Bloom filter checking. Second, as one secondary index query may need to search for multiple independent records in the primary table, we parallelize these searching operations with multiple threads. Since search latencies on the LSM-based primary table may vary largely, we apply a worker-active manner on parallel threads to avoid load imbalance among threads and improve utilization.

We implement PERSEID and evaluate it against state-of-the-art PM-based indexes and LSM-based secondary indexing techniques on PM. The evaluation results show that PERSEID outperforms exiting PM-based indexes by 3-7× for queries, and achieves about two orders of magnitude higher performance of state-of-the-art LSM-based secondary indexing techniques even if on PM instead of disks, while maintaining the high write performance of LSM-based storage systems.

In summary, this paper makes the following contributions:

- Analysis of the inefficiencies of LSM-based secondary indexing techniques and existing PM-based indexes when adopted as secondary indexes for LSM-based KV stores.
- PERSEID, an efficient PM-based secondary indexing mechanism, which includes a secondary index-friendly structure, a lightweight validation approach, and two optimizations on primary table searching issued from secondary indexes.
- Experiments that demonstrate the advantage of PERSEID.

2 Background

2.1 Log-Structured Merge Trees

The LSM-tree applies out-of-place updates and performs sequential writes, which achieves superior write performance compared to other in-place-update storage structures.

The LSM-tree has a multi-level structure on storage and each level comprises one or several sorted runs. The size of Level L_n is several times (e.g., 10) larger than Level L_{n-1}. Each sorted run contains sorted KV pairs and is further partitioned to multiple small components called *SSTables*. In LSM-trees, new key-value pairs are first buffered into a memory component called a *MemTable*. When the MemTable fills up, it turns into an immutable MemTable and gets flushed to storage as a sorted run. Since sorted runs have overlapping key ranges, a query operation needs to search multiple sorted runs. To limit the number of sorted runs and improve search efficiency, LSM-trees conduct compaction periodically to merge several components and remove obsolete KV pairs.

Two typical compaction strategy and their variants are commonly used in LSM-trees [19, 41]: The leveling strategy [22, 24] only allows each level (besides L_0) to have only one sorted run; The tiering strategy [48, 54] allows each level (besides L_0) to have multiple sorted runs to reduce the write amplification. Compared with leveling strategy, tiering strategy has a much smaller write amplification ratio and thus

[2] Index-only query is a common query technique: Users create a *covering index* that contains specific columns required by queries to avoid the cost of reading the primary table [5, 7, 44]. A non-index-only query searches the secondary index by secondary key to get primary keys and then retrieves full records from the primary table.

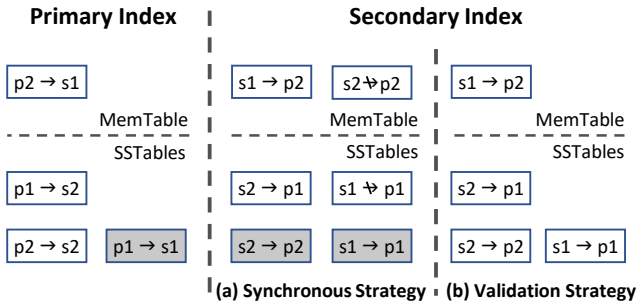


Figure 1: Stand-alone secondary indexing in LSM-based systems with Synchronous strategy and Validation strategy [47]. *The shaded entries indicate that they are invisible in the index.*

higher write performance. However, since query operations need to search multiple sorted runs in each level, LSM-trees with tiering strategy have much lower read performance.

2.2 Secondary Index in LSM-based Systems

Many applications require queries on specific values other than primary keys. Without an index based on specific values, database systems need to scan the whole table to find relevant data. Thus, secondary indexing is an indispensable technique in database systems. For example, in Facebook's database service for social graphs, secondary keys are heavily used, such as finding IDs who liked a specific photo [11, 44]. In this work, we mainly discuss stand-alone secondary indexes, which are separate index structures apart from the primary table and are commonly used in database systems [47]. A stand-alone secondary index maintains mappings from each secondary key to its associated primary keys. As secondary keys are not unique, a single secondary key can have multiple associated primary keys.

Consistency strategy. Since LSM-based KV stores update or delete records by out-of-place blind-writes, maintaining consistency of secondary indexes becomes a challenge in LSM-based storage systems. There are two strategies to handle this issue, *Synchronous* and *Validation*.

For *Synchronous* strategy, whenever a record is written in the primary table, the secondary index is maintained synchronously to reflect the latest and valid status (e.g., AsterixDB [9], MongoDB [4], MyRocks [44]). For example, as shown in Figure 1(a), when writing a new record { $p2 \rightarrow s1$ } (p denotes the primary key, s denotes the secondary key, and other fields are omitted for simplicity) into the primary table, the storage system also fetches the old record of $p2$ to get its old secondary key $s2$. Then the storage system inserts not only a new entry { $s1 \rightarrow p2$ } but also a tombstone to delete the obsolete entry { $s2 \rightarrow p2$ } in the secondary index. Nevertheless, this strategy discards the blind-write attribute and thus degrades the write performance which is the main advantage of LSM-based KV stores.

By contrast, as shown in Figure 1(b), *Validation* strategy only inserts the new entry { $s1 \rightarrow p2$ } but does not maintain the consistency of obsolete entries in secondary indexes (e.g., DELI [50], and secondary indexing proposed by Luo et al. [40]). However, secondary index query operations need to validate all relevant entries by checking the primary table to filter out obsolete mappings. Though previous work proposed some approaches to reduce the validation overhead, their benefits are limited. For example, DELI [50] lazily repairs the secondary index along with compaction of the primary table. Luo et al. [40] propose to store an extra timestamp for each entry in the secondary index and use a *primary key index* that only stores primary keys and their latest timestamp for validation. The primary key index is validated instead of the primary table. However, since the primary key index is also an LSM-tree, though it filters out unnecessary point lookups on the primary table, it still requires point lookups on itself.

Index type. As a secondary key can have multiple associated primary keys, LSM-based secondary indexes have two types surrounding this issue, including *composite index* and *posting list* [47]. The key in a *composite index* (i.e., *composite key*) is a concatenation of a secondary key and a primary key. The composite index is easy to implement and adopted by many applications [16, 44, 47]. However, it turns a secondary lookup operation into a prefix range search operation.

The *posting list* stores multiple associated primary keys in the value of a KV pair. When a new record is inserted, there are two update strategies. *Eager* update strategy conducts read-modify-write, fetching the old posting list and merging the new primary key to the posting list. *Lazy* update strategy blindly insert a new posting list which only includes the new primary key. It leaves posting lists merging to compaction. However, a secondary lookup needs to search all levels to fetch all relevant entries.

Limitations. Even though there are multiple strategies, types, and optimizations, LSM-based secondary indexes have to sacrifice either the write performance of storage systems or the secondary index query performance, which results from the incompatibility of inherent attributes of LSM-trees and characteristics of secondary indexes.

2.3 Persistent Memory

Persistent Memory (PM), also called Non-Volatile Memory (NVM) or Storage Class Memory (SCM), provides several attractive benefits for storage systems, such as byte-addressability, DRAM-comparable access latency, and data persistency. CPUs can access data on PM directly with load and store instructions. Compared to DRAM, PM has a much larger capacity and lower cost and power consumption. In addition to DDR bus-connected PM (e.g., Intel Optane DCPMM), the recent high-bandwidth and low-latency IO interconnection, Compute Express Link (CXL) [3, 29], brings a new form of SCM, CXL device-attached memory (e.g., Samsung's Memory-semantic SSD [8]).

However, PM also has some performance idiosyncrasies.

For example, the current commercial PM hardware (i.e., Intel Optane DCPMM) has physical media access granularity of 256 bytes, leading to high random access latency (about 3× of DRAM) and write amplification for small random writes, which needs to be considered when designing PM systems [14, 51, 53, 55, 57, 60]. These idiosyncrasies should be more obvious on CXL device-attached memory due to the physical media characteristics (e.g., flash page in CXL-SSD).

3 Motivation

Though recent work introduces some techniques to optimize secondary indexing in LSM-based systems, we find that *the performance of LSM-based secondary indexing is still unsatisfactory due to the incompatibility of inherent attributes of LSM-trees and characteristics of secondary indexing*. On the one hand, LSM-tree is not a competent data structure for secondary indexes, since the characteristics of secondary indexes exacerbate the deficiency of LSM-tree's read operations: (1) KV pairs are usually small in secondary indexes, to which LSM-tree's cumbersome lookup operations are unfriendly; (2) Secondary keys are not unique and can have multiple values, which LSM-tree's out-of-place update will exacerbate the query inefficiency. On the other hand, the blind-write attribute of LSM-based primary tables makes the consistency of secondary indexes troublesome.

Therefore, this motivates us to find a better solution for secondary indexes in LSM-based storage systems. As PM provides attractive features such as byte-addressability, DRAM-comparable access latency, and data persistency, we argue that it is promising to provide secondary indexing with PM.

Though there are many state-of-the-art PM-based index structures, they are not specifically designed for secondary indexes. To adopt them as secondary indexes (e.g., support the multi-value feature), naive approaches include the composite index or using a conventional allocator to organize posting lists (§2.2). However, *simply adopting these naive approaches to use existing PM-based indexes as secondary indexes will overshadow their superior advantages*.

Why not use a PM-based composite index? Though this method is straightforward and easy to implement in LSM-based systems, it is not ideal for tree-based persistent indexes. First, when adding or removing a primary key for a secondary key, a value update operation turns into a new composite key insert or delete operation for composite indexes. Insert and delete operations are more expensive than update operations in a PM-based tree index because they may cause shift operations or structural modification operations (SMO). Second, composite indexes store every pair of mappings as an individual KV pair, expanding the number of KV pairs, which increases the height of the tree index and thus degrades its query performance. Third, storing the same secondary keys repeatedly in multiple composite keys wastes PM space, which can be a dominant overhead for some real-world databases [59].

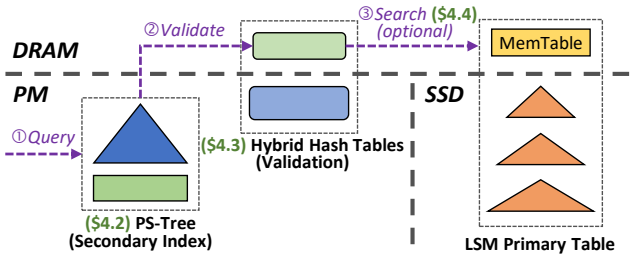


Figure 2: The overall architecture with Perseid.

Why not use a conventional allocator? One may use a conventional allocator, such as a log-structured approach or a slab-based allocator, to allocate space for values out of the indexes. Nevertheless, they are not suitable for values of secondary keys. The log-structured approach is friendly to PM for its sequential-write pattern. One may use the log-structured approach to append new values in the log and use pointers to link associated values belonging to the same secondary key. However, it will scatter values (primary keys) associated with the same secondary key and thus reduce the query performance due to poor data locality. Slab-based allocators are widely used for volatile memory, but they are not suitable for PM and secondary indexes. First, these general-purpose allocators usually have high overheads on PM since they conduct expensive mechanisms for crash consistency (e.g., logging) and perform many small writes on their metadata which is necessary for recovery [6]. Second, slab-based allocators have low memory utilization due to the memory fragmentation issue [49], which cannot be eliminated by restarting on PM [18]. These issues are more severe for secondary indexes. In secondary indexes, the value of a secondary key is changed by inserting or removing primary keys, which means the value size (the total size of associated primary keys) changes constantly. This characteristic not only needs frequent reallocation but also exacerbates the fragmentation issue.

Our experiments (§5.2) show that these naive approaches on PM-based indexes lead to several times performance degradation. It thus motivates us to explore a new PM-based secondary indexing mechanism for LSM-based KV stores. In addition, an efficient validation approach is required to retain the blind-write attribute of LSM-based KV stores.

4 Perseid Design

4.1 Overview

Motivated by the analysis above, we propose PERSEID, a PM-based secondary indexing mechanism for LSM-based storage systems, which overcomes traditional LSM-based secondary indexes' deficiencies. Figure 2 shows the overall architecture of an LSM-based storage system with PERSEID.

- PERSEID contains a PM-based secondary index, `PS-Tree`, which is both PM and secondary index friendly: by adopting

log-structured insertion, PS-Tree achieves fast insertion on PM; by storing primary keys which associate to the same secondary key closer and further rearranging them adjacent, PS-Tree supports efficient query operations (§4.2).

- PERSEID retains the blind-write attribute of the LSM primary table for write performance without sacrificing query performance by introducing a lightweight hybrid PM-DRAM and hash-based validation approach. The validation approach contains a persistent hash table to record version information of primary keys, and a volatile and lite hash table to absorb random accesses to PM. (§4.3).
- To accelerate non-index-only queries, PERSEID adapts two optimizations on primary table searching issued from secondary indexes. PERSEID filters out irrelevant component searching with sequence numbers and parallelizes primary table searching in an efficient way (§4.4).

4.2 PS-Tree Design

PERSEID introduces PS-Tree, a PM-based secondary index, which is designed considering the multi-value feature and PM characteristics. We first present PS-Tree's structure (§4.2.1), and then describe its operations (§4.2.2).

4.2.1 Structure

The overall structure of PS-Tree is shown in Figure 3. PS-Tree consists of two layers, *SKey Layer* for indexing secondary keys and *PKey Layer* for storing values. Specifically, the SKey Layer resembles a normal in-memory index and can leverage an existing high-performance PM-based index (e.g., P-Masstree [33, 43] and FAST&FAIR [25]). The PKey Layer stores variable-number values (i.e., primary keys and other user-specified values) of secondary keys in a manner of blended B$^+$-Tree leaf nodes and log-structured approaches, which combines the advantages of the two approaches. The value of a secondary key in the SKey Layer is a pointer, which points to corresponding primary keys in the PKey Layer. Each pointer is a combination of the address of the PKey Page and an offset within the page.

In the PKey Layer, primary key entries (*PKey Entries*) are stored in *PKey Pages*. Each PKey entry has an 8-byte metadata header and a primary key. The header consists of a 2-byte size, a 1-bit obsolete flag, and a 47-bit sequence number (SQN) of the primary key. The SQN is internally used for multi-version concurrency control (MVCC) in LSM-based KV stores [22, 24]. Each new record (including updates and deletes) in the primary table gets a monotonically increased SQN. PERSEID leverages the SQN mechanism to guarantee data consistency among the primary table and secondary indexes, and also for validation which will be described in §4.3. PKey Pages are aligned to PM physical media access granularity (e.g., 256 bytes of Intel Optane DCPMM [57]). PS-Tree inserts PKey Entries into PKey Pages in a log-structured manner to reduce the write overhead and ease crash consistency on PM.

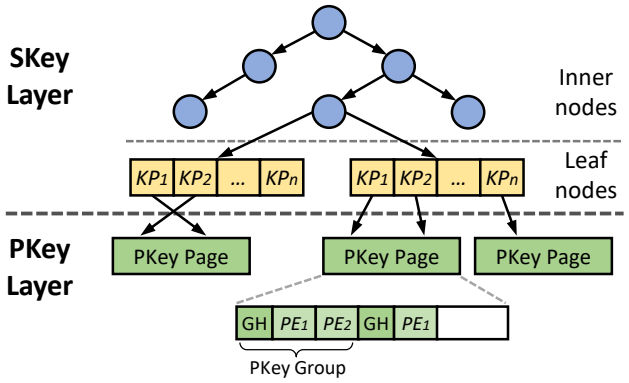


Figure 3: The structure of PS-Tree. *KP: Key Pointer pair, GH: Group Header, and PE: PKey Entry.*

Nevertheless, traditional log-structured approaches scatter different values of the same secondary key in the log, resulting in poor data locality and degraded query performance. To improve data locality, PS-Tree stores PKey Entries of contiguous SKeys in the same PKey Page, similar to the leaf node in a B$^+$-Tree. Furthermore, during the PKey Page split, PS-Tree rearranges PKey Entries that belong to the same secondary keys to store continuously as a *PKey Group*. Each PKey Group has an 8-byte *Group Header* and one or multiple PKey Entries. The lower 48 bits of a group header are the address of the previous PKey Group of the same secondary key or null if the current group is the last one. Thus all PKey Groups belonging to one secondary key are linked as a list. The remaining 16 bits store the number of total entries and the number of obsolete entries in the group.

4.2.2 Basic Operations

PS-Tree considers features of both secondary indexing and persistent memory. Compared with DRAM, PM has limited write bandwidth and the write amplification issue. Therefore, PS-Tree adopts log-structured insertion and copy-on-write split for efficient writes and lightweight crash consistency mechanisms. To avoid high latencies of multiple random accesses of multiple values on PM during query operations, PS-Tree reorganizes values of the secondary index and conducts lazy garbage collection during the PKey Page split.

Log-structured Insert. Algorithm 1 describes the process of the insert operation. First, PS-Tree searches for the SKey and its pointer in the SKey Layer. From the pointer, PS-Tree locates the previous PKey Group and the corresponding PKey Page (Line 1-3). If the SKey is not found, then the PKey Page is located from the pointer of the previous SKey which is just smaller than this new SKey (Line 5).

Second, PS-Tree appends a new PKey Group in that PKey Page (Line 11-12). The new PKey Group contains one entry with the new PKey and other values if specified, and the header points to the previous PKey Group if exists.

Third, the new pointer of the SKey (i.e., the address of

Algorithm 1: Insert(SKey sk, PKey pk, Slice val, SeqNumber seq)

```
1  search for the leaf_node and pointer ptr of sk in SKey Layer;
2  if ptr ≠ NULL then // found sk
3  |   pkey_page = pointer.pkey_page;
4  else
5  |   pkey_page = leaf_node→get_prev_pkey_page(sk);
6  end
7  if pkey_page is full then
8  |   pkey_page split;
9  |   goto Line 1;
10 end
11 construct a PKeyGroup pg with pk, val, seq, and ptr;
12 new_ptr = pkey_page→append(pg);
13 leaf_node→upsert(sk, new_ptr);
```

the new PKey Group) is updated or inserted in the SKey Layer (Line 13). Thus, the insert request usually performs an update operation in the SKey Layer. PKey Entries of a secondary key are always linked in the order of recency to facilitate query operations which usually require the most latest entries [11, 47].

Search. The search operation in PS-Tree starts with searching for the secondary key and its pointer in the SKey Layer. Then, from the latest PKey Group indicated by the pointer, primary keys and other user-specified values can be retrieved in the order of recency. PERSEID adopts the *Validation* strategy (§2.2) for its high ingestion performance. Therefore, all primary keys are first validated before returning. Obsolete PKey Entries are marked as obsolete by setting their obsolete flags, which will be removed physically when the PKey Page splits. The LSM-based primary table supports MVCC by attaching one snapshot and using reference counters to protect components from being deleted [22, 24]. In PS-Tree, we adopt an epoch-based approach: readers publish their snapshot numbers during query operations, and obsolete entries whose sequence number is larger than any reader's snapshot number are guaranteed not to be removed physically.

Update and Delete. PS-Tree has no update or delete operations (from the point of view of secondary indexes rather than the data structure). Since updating the primary key of a record in the primary table is commonly not supported in database systems, there is no need to update values (primary keys) in secondary indexes. With the Validation strategy, PS-Tree does not delete the obsolete entries synchronously with the primary table. PS-Tree leaves obsolete entry cleaning to garbage collection.

Locality-aware PKey Page Split with Garbage Collection. When a PKey Page does not have enough space for a new entry, it splits into two new PKey Page in a copy-on-write manner. Since insertions are performed in a log-structured manner, the PKey Entries which belong to one SKey may

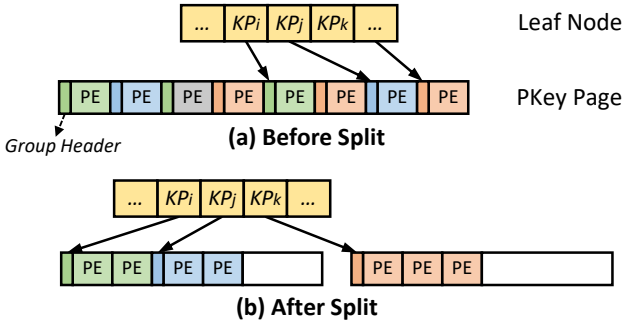


Figure 4: An example of PKey Page split. *PEs (PKey Entries) with the same color belong to the same secondary key; PE in gray are obsolete.*

scatter discontinuously. Querying these entries may need multiple random accesses on PM. As PM has non-negligible read latencies compared to DRAM (e.g., about 300 ns with Intel Optane DCPMM [57]), query operations can have high overheads. Therefore, as shown in Figure 4, to improve locality, PS-Tree reorganizes PKey Entries when the PKey Page splits. PS-Tree rearranges PKey Entries belonging to the same SKey in one PKey Group, so these entries are stored continuously, and the storage overhead of the Group Header is reduced since multiple PKey Entries share one Group Header.

Besides, entries not marked as obsolete in the current PKey Page are validated by a lightweight approach (described in §4.3) and obsolete entries are physically removed during reorganization to reduce space overhead. Since a secondary key may have many primary keys which occupy more than one PKey Page, for those PKey Entries not in the current PKey Page, PS-Tree lazily garbage collects them only when the number of obsolete entries exceeds half of the number of total entries in that PKey Group. To support MVCC, PS-Tree retains obsolete entries whose sequence number is larger than the minimum snapshot number of concurrent readers. Obsolete entries may retain long if there exists a long-running queries. PERSEID can be enhanced with similar techniques in recent work [30, 35] to handle long-lived snapshots. After rearranging valid entries to the new PKey Pages, pointers of related SKeys are updated and the old PKey Page is freed.

Crash Consistency. PERSEID relies on the existing write-ahead-log (WAL) of the LSM-based primary table to guarantee atomic durability among the primary table and secondary indexes. During recovery with WAL, PERSEID redoes uncompleted operation to the PS-Tree.

PS-Tree also handles its own crash consistency issues. Insert operations are committed only when the pointers in SKey Layer are updated. If the system crashes before updating pointers but after allocating a new PKey Page, then the PKey Page is unreachable. After restart, a background thread will scan the allocated pages and PS-Tree to find and reclaim unreachable pages. Besides, PS-Tree allows concur-

rent insertion in one PKey Page. A thread obtains the space to write by compare-and-swap the tail pointer of the PKey Page. Thus, the space may leak if any thread obtains it but does not update the pointer in SKey Layer when the system crashes. `PS-Tree` tolerates this situation and leaves these leakages to page splitting which naturally reclaims these leaked spaces.

4.3 Hybrid PM-DRAM Validation

PERSEID adopts the Validation strategy (see §2.2) for high write performance, which necessitates a lightweight validation approach. Since update-intensive workloads are quite common nowadays [11, 12], if the validation approach is heavy, validating a large number of obsolete entries brings no outcomes but generates huge overhead.

PERSEID introduces a lightweight validation approach based on the requirement of validation. PERSEID adopts a hash table on PM storing version information for primary keys. The hash table is indexed by the primary key and stores its latest sequence number (§4.2.1). Nevertheless, even though point lookups on a PM-based hash table are much faster than on a tree, the validation time is comparable to the query time of `PS-Tree`. This is because one secondary key has multiple primary keys to validate, and PM has non-negligible random access latency. Simply placing the hash table on DRAM will occupy a large memory footprint. However, as validation only needs to validate whether a version of a primary key is valid, but not obtaining the specific latest version number, PERSEID builds another volatile hash table on DRAM which only stores versions for primary keys that have been updated or deleted. In this way, PERSEID only needs to query the small volatile hash table and thus the validation overhead is further reduced.

Figure 5 illustrates the hybrid PM-DRAM validation approach. The values in the hash tables consist of the sequence number of the record (6-byte) and a 2-byte counter. The counter is used to determine whether a primary key has obsolete versions. There is a slight difference in the counters of the two hash tables. In the volatile hash table, each counter indicates the number of *logically* existing entries related to a primary key in the secondary index. By contrast, each counter in the persistent hash table indicates the number of *physically* existing entries in the secondary index. Next, we describe the validation approach in detail according to operations.

Insert. When a new record (including update and delete) is inserted into the primary table, the primary key is inserted or updated with its sequence number into the persistent hash table. If the persistent hash table does not contain this primary key before, its counter is set to one, which means this primary key has only one version and no obsolete entries of this primary key exist in the secondary index. For example in Figure 5, at *t2*, key *c* is inserted for the first time, and it is inserted into the persistent hash table. Otherwise, the primary key's counter in the persistent hash table is increased by one; besides, the primary key is inserted or updated with its sequence number into the volatile hash table, and the counter

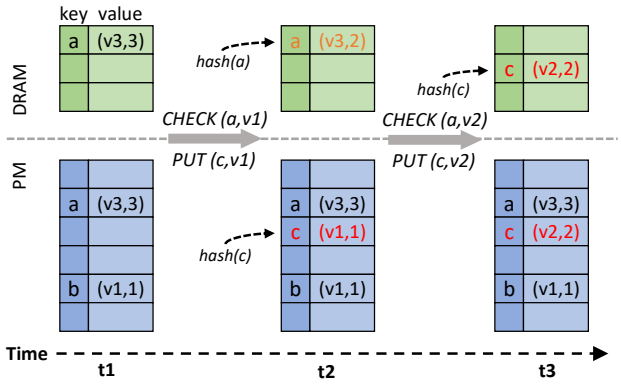


Figure 5: Hybrid PM-DRAM hash-based validation.

in the volatile hash table is set to two if it's an insertion or increased by one if it's an update. For example, when key *c* is updated with a new version *v2* at *t3* in Figure 5, the entry in the persistent hash table is updated and a new entry is inserted into the volatile hash table.

Validate. The secondary index validates an entry by querying the volatile hash table. Specifically, the entry is valid if the sequence number of this entry matches the latest sequence number stored in the hash table, or the hash table does not contain the primary key which means there are no obsolete entries of this primary key. Otherwise, the entry is not valid and PERSEID marks the entry as obsolete and decreases the counter of the entry in the volatile hash table by one. For example, when key *a* is checked with an obsolete version *v1* at *t2* in Figure 5, the result is false, and then the counter is decreased from 3 to 2. If the counter is decreased to 1, which means all obsolete entries have been marked, the entry is removed from the volatile hash table to restrict the hash table size. For example, when key *a* is checked with an obsolete version *v2* at *t3* in Figure 5, the counter is decreased to one, the validation return false and the entry is removed. A rare case is that the volatile hash table reports a new sequence number larger than the current reader's snapshot number, which means a concurrent writer has updated this primary key. In this case, we cannot directly confirm whether this entry is valid in this snapshot, so we have to validate it by the primary table. During validation for secondary index queries, PERSEID only operate with the volatile validation hash table. Thus, the validation overhead is quite small.

Garbage Collection. During the PKey Page split, entries that are not marked as obsolete are also validated to remove obsolete entries (§4.2.2). Since this step physically removes obsolete entries, PERSEID decreases the corresponding counters in the persistent hash table. If a counter is decreased to one, PERSEID removes the corresponding hash pairs from the volatile hash table.

Recovery. When the system restarts from a crash or a normal shutdown, the volatile hash table needs to be recovered. PERSEID iterates the whole persistent hash table and

inserts primary keys whose counter is greater than one into the volatile hash table. Now the counters in the volatile hash table are numbers of physically existing entries, which may be larger than the actual numbers of logically existing entries. Therefore, some false positive primary keys may exist in the volatile hash table. However, this does not affect the validation accuracy and these primary keys can be removed by garbage collection.

4.4 Non-Index-Only Query Optimizations

Though the PERSEID significantly reduces the overhead of secondary indexing, the overhead of non-index-only queries (require full records) is still dominated by the LSM-based primary table. Thus, PERSEID further introduces two optimizations for non-index-only queries.

4.4.1 Locating Components with Sequence Number

A secondary index query operation may need to search the primary LSM table multiple times for all its associated records. LSM-trees have mediocre read performance due to the multi-level structure. Besides device I/Os, if data is cached in memory or using fast storage devices, LSM-trees have non-negligible overheads on probing components (i.e., indexing and checking Bloom filters) [17, 20, 60]. Moreover, the read performance gets worse with the tiering compaction strategy since more components (SSTables) need to check and read.

Nevertheless, we find that many components are unnecessary to probe in searching processes issued from the secondary index. Previous work uses zone maps, which store the minimum and maximum values of an attribute, to skip irrelevant data blocks or components during searching [9, 10, 47]. We found that this technique can also be used by secondary indexes to search the primary table. Since we have already recorded the sequence numbers of primary keys in the secondary index, the sequence number can be used as an additional attribute to skip irrelevant components. PERSEID builds a zone map that records a sequence number range (i.e., the minimum and maximum sequence numbers of records) for each component (including MemTables).

Moreover, since tiering compaction does not rewrite SSTables in the higher level (except for the last level), for a range partition, the sequence number ranges of different levels and even different sorted runs are strictly divided. For primary tables adopting the tiering strategy, with the primary key to search SSTables horizontally and the additional sequence number to search sorted runs vertically, PERSEID can locate the exact component that contains the record directly. Besides, since PERSEID already validates the version so it must exist in the component, PERSEID can further skip the Bloom filter checking. Thus, the overheads on indexing and checking Bloom filters are almost eliminated.

This optimization does not fit with leveling strategy perfectly. The sequence number ranges in different levels have overlaps because compaction rewrites SSTables in the higher level and generates new SSTables with blended sequence numbers from multiple levels. However, since most LSM-base KV stores adopt the tiering strategy on L_0 at least [22, 24], this optimization is still effective to some extent.

4.4.2 Parallel Primary Table Searching

A single secondary key usually has multiple associated primary keys, and queries on these primary keys are independent. Therefore, using multiple threads to accelerate primary table searching is a natural optimization method. One naive approach is to assign primary keys to each thread in a round-robin fashion. However, point lookups on LSM-trees may have a large latency gap, since some KV pairs can be fetched from MemTable or block cache directly and others may reside at a relatively high level and need several disk I/Os due to Bloom filter false positives. The naive approach will result in a load imbalance among parallel threads where some threads have finished their tasks and become idle while others are still stuck and there may still exist some unfinished tasks.

To avoid this situation, we apply a *worker-active* fashion. PERSEID publishes primary keys into a shared queue as tasks, and each parallel thread fetches one task from the queue. In this way, though each thread may complete a different number of tasks, parallel threads are fully utilized.

5 Evaluation

In this section, we evaluate PERSEID against existing PM-based indexes with naive approaches and state-of-the-art LSM-based secondary indexing techniques [40, 47]. After describing the experimental setup (§5.1), we evaluate these secondary indexing mechanisms with micro benchmarks to show their performance on basic operations (§5.2). Then, we evaluate these systems' overall performance with mixed workloads (§5.3) and recovery time (§5.4).

5.1 Experimental Setup

Platform. Our experiments are conducted on a server with an 18-core Intel Xeon Gold 5220 CPU, which runs Ubuntu 20.04 LTS with Linux 5.4. The system is equipped with 64 GB DRAM, two 128 GB Intel Optane DC Persistent Memory in AppDirect mode, and a 480 GB Intel Optane 905P SSD.

Compared Systems. We implement `PS-Tree` of PERSEID based on two typical PM-based indexes, a B$^+$-Tree FAST&FAIR [25] and a trie-like P-Masstree [33, 43]. We compare PERSEID against the two original PM-based indexes, and LSM-based secondary index (denoted as `LSMSI`) approaches of *LevelDB++* [47]. The compared PM-based indexes are implemented as secondary indexes via the composite index approach and the log-structured approach (denoted as `FAST&FAIR-composite`, `FAST&FAIR-log`, `P-Masstree-composite`, `P-Masstree-log`, respectively). For the log-structured approach, we provide enough space

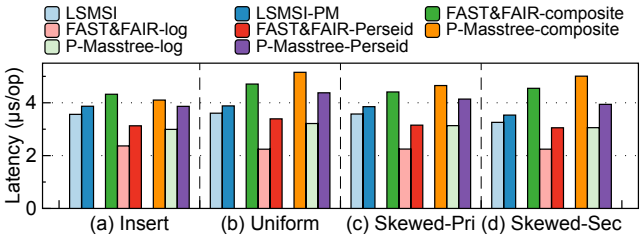


Figure 6: Insert and upsert performance.

and disable garbage collection to avoid its influence and present the ideal performance [51]. We enhance other PM-based indexes with PERSEID's hybrid PM-DRAM validation approach (§4.3) and `LSMSI` with the primary key index [40] (§2.2) for validation. For a fair comparison, we also implement LSM-based secondary indexing approaches on PM. We use *PebblesDB* [48], a state-of-the-art tiering-based KV store, as the primary table.

Workloads. Since common benchmarks for key-value stores such as YCSB [15] don't have operations on secondary indexes, as in previous work [36, 40, 47], we implemented a secondary index workload generator based on an open-source twitter-like workload generator [2] for evaluation. With this generator, we generate several microbenchmark workloads and mixed workloads. The primary key (e.g., ID) and secondary key (e.g., UserID) are randomly generated 64-bit integers. The key space of primary keys and secondary keys is 100 million and 4 million, respectively. Thus the average number of records per secondary key is about 25. The size of each record is 1KB.

KV Store Configurations. For the primary table, according to configuration tuning guide [23], MemTable size is set to 64 MB and the Bloom filters are set to 10 bits per key. As our workloads will generate a primary table larger than 100 GB, we set a 16-GB block cache for the primary table and a 1-GB block cache for the LSM-based secondary index. Compression is turned off to reduce other influencing factors.

5.2 Microbenchmarks

In this section, we evaluate the basic single-threaded performance and scalability of compared secondary indexing mechanisms.

5.2.1 Insert and Update

The *Insert* workload (i.e., no updates) has 100 million unique records. Figure 6(a) shows the average latency of insert operations of each secondary index. PERSEID performs about 10-38% faster than the corresponding composite indexes, but 25% slower than the *ideal* log-structured approach without garbage collection due to the page split overhead in `PS-Tree`. The composite index approach results in inferior performance as we analyzed in §3. Other approaches have higher performance due to the sequential-write pattern.

The upsert workloads contain 100 million insert operations and 100 million update operations. Operations are shuffled to avoid all newer entries being valid in secondary indexes. In the *Uniform* workload, both primary keys and secondary keys follow a uniform distribution. In the *Skewed-Pri* workload, primary keys follow a Zipfian distribution with the skewness parameter 0.99, and secondary keys are selected randomly. In the *Skewed-Sec* workload, secondary keys follow a Zipfian distribution (parameter 0.99), and primary keys are uniform. Thus, hot secondary keys have lots of associated primary keys, which represent low-cardinality columns.

Compared with other secondary indexes, composite indexes perform even worse in upsert workloads. This is because, with additional upsert operations, composite indexes have more KV pairs and larger tree heights. By contrast, `PS-Tree` and the log-structured approach do not increase the number of KV pairs in the index part.

5.2.2 Query

In this experiment, we evaluate the performance of index-only queries of each system after loading the insert workload or upsert workloads. Index-only query reflects the performance of a secondary index itself and is a common query technique (i.e., covering index [5, 7]) to avoid looking up the primary table. We show two different selectivities by specifying limit N (10 and 200) on return results. The most recent and valid N entries are returned. For limit of 200, the actual average number of returned entries per query is 25 and 142 for the *Skewed-Pri* and *Skewed-Sec*, respectively.

Figure 7 shows the results of index-only query performance. From the results, we have the following observations.

First, PM-based indexes have significantly lower latencies than LSM-based secondary indexes. Putting `LSMSI` on PM (`LSMSI-PM`) has very limited improvement, which is because `LSMSI` already benefits from block cache and OS page cache. Even so, `LSMSI` is still inefficient due to the high overhead of indexing and Bloom filter checking. Besides, `LSMSI` has a high overhead on validating the primary key index.

Second, PERSEID outperforms existing PM-based indexes with the composite index and the log-structured approach by up to 4.5× and 4.3×, respectively. The log-structured approach has poor locality since relevant values are scattered across the whole log and require multiple random accesses to fetch them all. Composite indexes are inferior due to the larger number of KV pairs in the indexes and range-scan operations as we analyzed in §3. They are especially inefficient under the *Skewed-Sec* workload with a large limit (e.g., 200), where they fetch a large number of entries and fail to enjoy the cache effect. By contrast, the performance of PERSEID is much more stable across different workloads, owing to the locality-aware design of `PS-Tree`. For a limit of 10, PM-based secondary indexes benefit from higher cache hit ratios under the *Skewed-Sec* workload, thus achieving better performance than other upsert workloads. Composite indexes also occupy about 4×

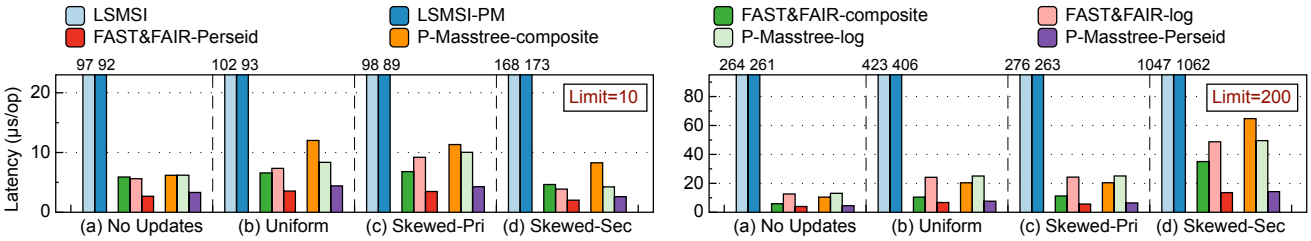


Figure 7: Index-only query performance.

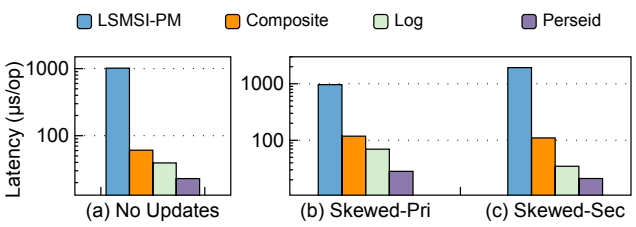


Figure 8: Index-only range query performance.

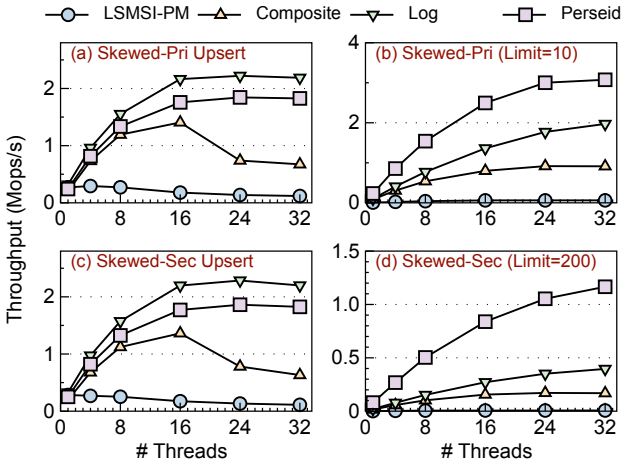


Figure 9: Multi-threaded performance.

more PM space than `PS-Tree`, which is because they repeatedly store secondary keys and have more index nodes. In addition, `P-Masstree-composite` has higher latencies than `FAST&FAIR-composite`, because trie-based indexes are less efficient than B$^+$-Trees in range search since their leaf nodes do not have sibling pointers pointing to neighbor nodes.

Third, under upsert workloads, all systems need to validate more primary keys to exclude obsolete entries, which also contributes to the higher overheads than under insert workloads. For `LSMSI`, since the primary key index needs multiple heavy point lookups on LSM-trees, validating the primary key index accounts for the lion's share of the total cost of an index-only query. `LSMSI` has lower latencies under the *Skewed-Pri* workload than other upsert workloads since the primary key index enjoys the data locality on primary keys. By contrast, PERSEID (and other PM-based indexes) validates on a volatile hash table, which takes up less than half of the total cost. The overhead on PERSEID increased little due to the locality-aware design of `PS-Tree` and the lightweight validation approach.

5.2.3 Range Query

In the following experiments, we show results of the LSM-based secondary index on PM (`LSMSI-PM`), and PM-based secondary indexes based on P-Masstree as representatives. We evaluate the range queries of these secondary indexes. Each range query searches for 20 secondary keys and retrieves 5 latest associated primary keys of each secondary key.

The results are shown in Figure 8. Range queries need to search more KV pairs from ten different secondary keys, showing a more pronounced difference between these secondary indexes than low-limit query operations. PERSEID outperforms `LSMSI-PM`, the Composite P-Masstree, and the log-structured approach by up to 92×, 5.2×, and 1.6×, respectively under the *Skewed-Sec* workload. Though `LSMSI-PM` benefits from PM access latency and DRAM caching, it still has a fairly high latency. This is because the range operation in LSM needs to merge-sort multiple iterators of components. The composite index needs to perform more range search than PERSEID in the index since PERSEID groups primary keys outside of the index.

5.2.4 Multi-threaded Performance

Figure 9 shows the multi-threaded performance of compared secondary indexes. We take the results of *Skewed-Pri* and *Skewed-Sec* workloads as representatives. For *Skewed-Sec*, we show the result with the limit of 200, and the result with the limit of 10 is similar to that of *Skewed-Pri*. For upsert operations, PERSEID scales up to 24 threads, achieving 2.8× and 16× the upsert throughput of the composite P-Masstree and `LSMSI-PM`, and slightly slower than the ideal log-structured approach. For query operations, PERSEID scales well and achieves 7× and 3× query throughput of `P-Masstree-composite` and `P-Masstree-log` under the *Skewed-Sec* workload due to the locality-aware design of `PS-Tree`. `LSMSI` has poor scalability due to its coarse-granularity lock and non-concurrent logging mechanism. Though using the same index structure (P-Masstree), because

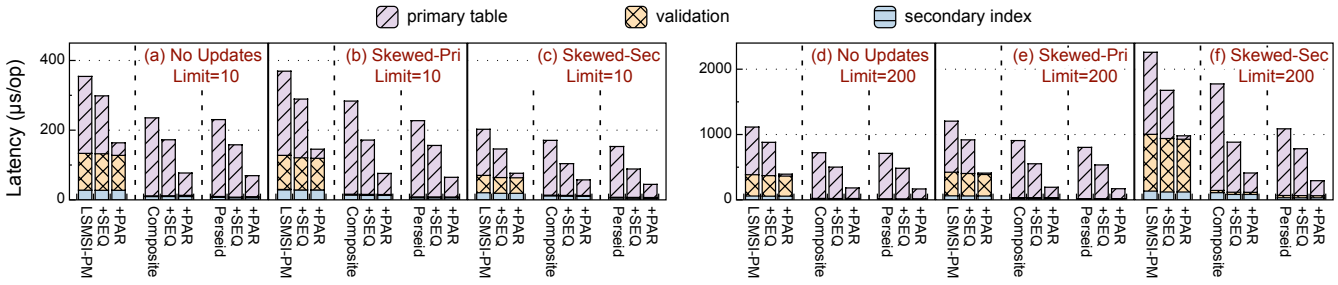


Figure 10: Non-index-only query performance. *The primary table time on +PAR only shows the time not covered by other parts.*

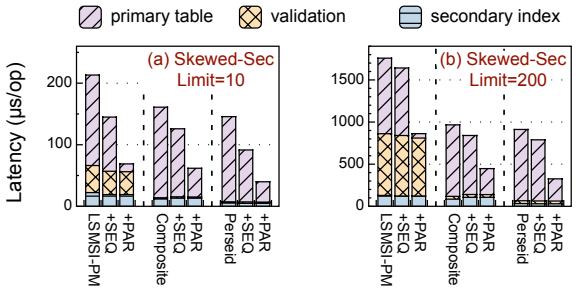


Figure 11: Non-index-only query performance on Leveling-based LSM table.

the composite index turns update operations into insert operations, the index operations limit its write scalability; and because it expands the number of KV pairs and thus has a bigger tree height, the increased index overhead limits its query scalability. As for the log-structured approach, the poor data locality restricts its query performance, especially for large-range queries.

5.2.5 Non-Index-Only Query

We next evaluate the non-index-only query operations. Besides the basic compared secondary indexes, we also enhance them by applying the two optimizations (§4.4), sequence number zone map (*+SEQ*) and parallel primary table searching (*+PAR*) sequentially. In this experiment, we use 4 threads for parallel primary table searching. Figure 10 shows the performance and time breakdown of non-index-only query operations. Note that the breakdown of primary table time on *+PAR* only shows the time not covered by the secondary index and validation. PERSEID brings considerable improvements against the `LSMSI-PM`, even if it has the two optimizations applied. PERSEID outperforms `LSMSI-PM` by up to 62% and 2.3×, when without and with optimizations on primary table lookups (the sequence number zone map and parallel primary table searching), respectively.

Though the primary key index indeed reduces unnecessary point lookup operations on the primary table for `LSMSI-PM`, with advanced low-latency storage devices and sufficient DRAM caching, it also has significant overhead. On the contrary, the hybrid PM-DRAM validation of PERSEID reduces the primary table lookups with subtle extra overhead.

PERSEID's optimizations on primary table searching can also boost other compared secondary indexing. The zone map improves the overall query performance of the KV store with PERSEID by about 50%, and the parallel primary table searching further improves by up to 3.1×. However, the numbers are only 20-36% and up to 2.4× for `LSMSI-PM`, respectively. This is because these optimizations only accelerate the primary table lookups, but the `LSMSI-PM` still has huge overheads. In addition, `LSMSI-PM` has to conduct the heavy validation first then it can pass the lookup tasks to parallel worker threads. Therefore, parallel threads cannot work adequately for `LSMSI-PM`.

We also implement secondary indexes and conduct the experiments on a leveling-based LSM primary table (LevelDB [24]). Figure 11 shows the results of *Skewed-Sec* as an example. The main difference is that the sequence number zone map is less effective on leveling-based LSM primary tables. However, the zone map is still effective when the limit is small, since the latest few records stay in MemTables or SSTables in lower levels like L_0, and these components can be filtered by sequence number with a high probability.

5.3 Mixed Workloads

Workload	Operation Ratios			
	Upsert	Get	Index-Only Query	Non-Index-Only Query
Write-Heavy	70%	10%	10%	10%
Balanced	45%	10%	25%	20%
Read-Heavy	20%	20%	40%	20%

Table 1: Mixed workloads description.

In this section, we evaluate PERSEID, the composite P-Masstree, and `LSMSI-PM` under mixed workloads. The mixed workloads consist of interleaved and various types of operations, which are more representative of real-world workloads. Each workload has 40 million operations, containing both *Skewed-Pri* and *Skewed-Sec* operations. Table 1 describes these workloads' traits. Each system is prefilled with 80 million records before performing the workloads. We also enable PERSEID's optimizations on primary table searching (i.e.,

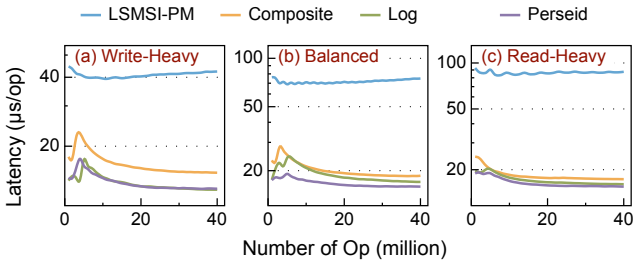


Figure 12: Performance of mixed workloads.

the sequence number zone map and parallel primary table searching) for all systems.

Figure 12 reports the average operation latencies every million operations. At the beginning of the Write-Heavy workload and the Balanced workload, PM-based secondary indexes have a spike in latency, which is mainly caused by seek-driven compaction in the LSM primary table. PERSEID outperforms `LSMSI-PM` significantly under different mixed workloads. Even though the overhead of the primary table dominates the whole operations, PERSEID still has visible advantages against the other PM-based indexes. Note that `PS-Tree` has much less capacity overhead than the composite index. As we set the limit on return results to 10 for query operations, the log-structured approach is not affected too much by its poor data locality.

5.4 Recovery Time

We evaluate the recovery time of PERSEID and `LSMSI-PM` after the Zipfian upsert workload that contains 200 million upsert operations with a single thread. Since we only need to recover the volatile validation hash table in PERSEID, it takes 2.7 seconds to scan the persistent hash table and rebuild the volatile hash table. By contrast, it takes 2.3 seconds and 1.4 seconds to recover the LSM-based secondary index and the primary key index, respectively. Their recovery time is mainly spent on rebuilding MemTables from logs and varies with the size of MemTables.

6 Related Work

Secondary Indexing in LSM-based KV stores. Qader et al. [47] conduct a comparative study on secondary indexing techniques in LSM-based systems. They conclude and evaluate several common secondary indexing techniques, including filter-based embedded index, composite index, and posting list. DELI [50] proposes an index maintenance approach that defers expensive index repair to compaction of the primary table. Luo et al. [40] propose several techniques for LSM-based secondary indexes, improving data ingestion and query performance. However, their techniques are mainly saving random device I/Os for traditional disk devices which reduce random reads at the cost of more sequential reads. Based on key-value separation [39], SineKV [36] keeps both the primary index and secondary indexes pointing to the record values. Thus, secondary index queries can get records directly without searching the primary index. However, SineKV has to discard the blind-write attribute and maintain index consistency synchronously. Cuckoo Index [32] enhances the filter-based indexing with a cuckoo filter. However, as a filter-based index, Cuckoo Index does not support range queries.

Though there are many proposed optimizations, LSM-based secondary indexing is not efficient enough due to the nature of LSM-trees. In this work, we revisit the design of the secondary index with persistent memory.

PM-based indexes. There has been plenty of research on high-performance PM indexes [13, 25, 31, 33, 42, 45, 46, 52, 56, 61]. These general-purpose indexing are not directly competent for efficient secondary indexing.

Improving LSM-based KV stores with PM. There is a lot of work optimizing LSM-based KV stores with PM. NoveLSM [27] introduces a large mutable MemTable on PM to lower compaction frequency and avoid logging. SLM-DB [26] utilizes a B$^+$-Tree on PM to index KV pairs on disks; SSTables on disks are organized in a single level, which reduces the compaction requirements. MatrixKV [58] places level L_0 on PM and adopts fine-granularity and parallel column compaction to reduce write stalls in LSM-trees. Facebook redesigns the block cache on PM to reduce the DRAM usage and thus reduce the total cost of ownership (TCO) [21, 28]. Different from these efforts, this work revisits the secondary indexing for LSM-based KV stores with PM.

7 Conclusion

In this paper, we revisit secondary indexing in LSM-based storage systems with PM. We propose PERSEID, an efficient PM-based secondary indexing mechanism for LSM-based storage systems. PERSEID overcomes the deficiencies of traditional LSM-based secondary indexing and existing PM-based indexes with naive approaches. `PS-Tree` achieves much higher query performance than state-of-the-art LSM-based secondary indexing techniques and existing PM-based indexes without sacrificing the write performance of LSM-based storage systems. The prototype of PERSEID is open-source at https://github.com/thustorage/perseid.

Acknowledgements

We sincerely thank our shepherd Baptiste Lepers and the anonymous reviewers for their valuable feedback. This work is supported by National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. 61832011, U22B2023, 62022051), and Huawei.

References

[1] Apache cassandra. https://cassandra.apache.org/, 2022.

[2] Chirp: A Twitter-like workload generator. http://alumni.cs.ucr.edu/~ameno002/benchmark/, 2022.

[3] Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/, 2022.

[4] Mongodb. https://www.mongodb.com, 2022.

[5] MySQL Glossary for Covering Index. https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_covering_index, 2022.

[6] Persistent Memory Development Kit. https://pmem.io/pmdk/, 2022.

[7] PostgreSQL: Documentation: Index-Only Scans and Covering Indexes. https://www.postgresql.org/docs/current/indexes-index-only-scans.html, 2022.

[8] Samsung electronics unveils far-reaching, next-generation memory solutions at flash memory summit 2022. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022/, 2022.

[9] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source bdms. *Proc. VLDB Endow.*, 7(14):1905–1916, oct 2014.

[10] Sattam Alsubaiee, Michael J. Carey, and Chen Li. Lsm-based storage and indexing: An old idea with timely benefits. In *Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data*, GeoRich'15, page 1–6, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1185–1196, New York, NY, USA, 2013. Association for Computing Machinery.

[12] 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.

[13] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.*, 13(12):2634–2648, July 2020.

[14] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[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 '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] 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. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[17] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171. USENIX Association, November 2020.

[18] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. Nvalloc: Rethinking heap metadata management in persistent memory allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 115–127, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Niv Dayan and Moshe Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 365–378, New York, NY, USA, 2021. Association for Computing Machinery.

[21] 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*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Facebook. Rocksdb. https://rocksdb.org/, 2022.

[23] Facebook. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2022.

[24] Sanjay Ghemawat and Jeff Dean. Leveldb. https://github.com/google/leveldb, 2022.

[25] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.

[26] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, February 2019. USENIX Association.

[27] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[28] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 821–837. USENIX Association, July 2021.

[29] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Power-optimized deployment of key-value stores using storage class memory. *ACM Trans. Storage*, 18(2), mar 2022.

[30] Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. Rethink the scan in mvcc databases. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 938–950, New York, NY, USA, 2021. Association for Computing Machinery.

[31] 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 '21, page 424–439, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter Boncz, and David G. Andersen. Cuckoo index: A lightweight secondary index structure. *Proc. VLDB Endow.*, 13(13):3559–3572, sep 2020.

[33] 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 '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[34] 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, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[35] 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.

[36] Fei Li, Youyou Lu, Zhe Yang, and Jiwu Shu. Sinekv: Decoupled secondary indexing for lsm-based key-value stores. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1112–1122, 2020.

[37] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+trees: Optimizing persistent index performance on 3dxpoint memory. *Proc. VLDB Endow.*, 13(7):1078–1090, mar 2020.

[38] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.*, 13(10):1147–1161, April 2020.

[39] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association.

[40] Chen Luo and Michael J. Carey. Efficient data ingestion and query processing for lsm-based storage systems. *Proc. VLDB Endow.*, 12(5):531–543, jan 2019.

[41] Chen Luo and Michael J. Carey. Lsm-based storage techniques: A survey. *The VLDB Journal*, 29(1):393–418, jan 2020.

[42] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 1–16. USENIX Association, February 2021.

[43] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.*, 13(12):3217–3230, aug 2020.

[45] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

[46] 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 '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. A comparative study of secondary indexing techniques in lsm-based nosql databases. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 551–566, New York, NY, USA, 2018. Association for Computing Machinery.

[48] 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 '17, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, page 1–16, USA, 2014. USENIX Association.

[50] Yuzhe Tang, Arun Iyengar, Wei Tan, Liana Fong, Ling Liu, and Balaji Palanisamy. Deferred lightweight indexing for log-structured key-value stores. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 11–20, 2015.

[51] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An efficient compaction approach for Log-Structured Key-Value store on persistent memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 773–788, Carlsbad, CA, July 2022. USENIX Association.

[52] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box approach to NUMA-Aware persistent memory indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111. USENIX Association, July 2021.

[53] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. Replicating persistent memory key-value stores with efficient rdma abstraction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, July 2023.

[54] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82, Santa Clara, CA, July 2015. USENIX Association.

[55] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Minhui Xie, Youyou Lu, Qing Wang, Yangyang Feng, Jiaqiang Liu, Kai Ren, and Jiwu Shu. Petps: Supporting huge embedding models with persistent memory. *Proc. VLDB Endow.*, 16(5):1013–1022, jan 2023.

[57] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[58] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.

[59] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-preserving key compression for in-memory search trees. SIGMOD '20, page 1601–1615, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: Differential indexing for persistent memory. *Proc. VLDB Endow.*, 13(4):421–434, December 2019.

[62] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association.

Zhuque: Failure is Not an Option, it's an Exception

George Hodgkins*
University of Colorado, Boulder

Yi Xu*
University of California, San Diego

Steven Swanson
University of California, San Diego

Joseph Izraelevitz
University of Colorado, Boulder

Abstract

Persistent memory (PMEM) allows direct access to fast storage at byte granularity. Previously, processor caches backed by persistent memory were not persistent, complicating the design of persistent applications and reducing their performance. A new generation of systems with flush-on-fail semantics effectively offer persistent caches, offering the potential for much simpler, faster PMEM programming models.

This work proposes Whole Process Persistence (WPP), a new programming model for systems with persistent caches. In the WPP model, all process state is made persistent. On restart after power failure, this state is reloaded and execution resumes in an application-defined interrupt handler.

We also describe the Zhuque runtime, which transparently provides WPP by interposing on the C bindings for system calls in userspace. It requires little or no programmer effort to run applications on Zhuque.

Our measurements show that Zhuque outperforms state of the art PMEM libraries, demonstrating mean speedups across all benchmarks of 5.24× over PMDK, 3.01× over Mnemosyne, 5.43× over Atlas, and 4.11× over Clobber-NVM. More important, unlike existing systems, Zhuque places no restrictions on how applications implement concurrency, allowing us to run a newer version of Memcached on Zhuque and gain more than 7.5× throughput over the fastest existing persistent implementations.

1 Introduction

Persistent memory (PMEM) exposes fast storage devices as byte-addressable main memory, allowing the processor to access persistent data via load and store instructions. The durability of PMEM enables an application's in-memory data to survive across system reboots and unexpected power failures. It promises to realize a vision of high performance, data persistence, a simple programming interface, and low storage overhead at the same time.

*The first two authors contributed equally to this work

However, building a system that realizes the promise of persistent programming is not simple. The contents of CPU caches do not survive power loss, and, since caches may delay evicting a modified cache line, writes may not reach PMEM in program order. This makes reasoning about the state of memory after a crash extremely challenging.

Programming systems (e.g., libraries, programming models, language support, and compilers) to help address the challenges of persistent memory programming have proliferated over the last decade. Broadly, three families of systems have emerged: each takes a different approach to consistency, and each faces significant challenges which bar widespread adoption.

The first and largest family [50, 63, 68, 71] requires programmers to access persistent state only through well-defined atomic operations (often called transactions). This provides a clean notion of consistency: after recovery from crash, each atomic section has either executed entirely or not at all. However, like all transactional memory models, this approach suffers from serious weaknesses: it is fundamentally incompatible with non-transactional synchronization, and has never gained significant traction in real systems.

The second family of systems [6, 26, 30, 42] uses FASEs, regions of code protected by locks, as atomic regions for PMEM updates. Legacy code can run with minimal changes, but these systems suffer from fundamental weaknesses arising from complex locking schemes and external IO. As we will show, addressing these weaknesses either cripples the system or essentially reduces it to a transaction-based system.

The final family of systems takes the more dramatic step of making *everything* in the system persistent via whole-system persistence (WSP) [47]. WSP provides the conceptually simplest programming model: Nothing much changes and, from the program's perspective, crashes never occur. WSP faces two major challenges: First, making all of memory persistent has until recently been infeasible, because regularly flushing volatile caches to PMEM creates enormous performance overheads. Second, making *everything* persistent would require a far-reaching redesign of many system components, for an

unclear benefit.

We think that WSP-style persistence is due for a renaissance: The advent of PMEM devices and platforms supporting *flush-on-fail* semantics (e.g. eADR for NVDIMMs or GPF for CXL devices) allows developers to treat caches as effectively persistent [14, 29], removing the main performance argument against WSP. Further, we believe that limiting the scope of persistence to a process – yielding *Whole Process Persistence (WPP)* – and providing well-defined, application-level semantics for system failures combine to produce a programming model that is fast, flexible enough to support legacy programs and complex locking schemes, and easy for programmers to use and understand.

WPP provides a simple abstraction to the process: its entire memory is persistent and will survive a power outage. If a power outage occurs, the process receives an OS signal after restart notifying it of the crash. The process can install a normal error handler for this signal which cleans up and exits, or performs more complex application-specific recovery; by default, program execution simply continues at the point of failure.

This work makes the following contributions:

- We identify a fundamental limitation of FASE-based PMEM systems.
- We introduce the WPP programming model, which treats power failure as a recoverable exception.
- We build the Zhuque runtime which provides WPP and describe its design and implementation.
- We provide experiments demonstrating the viability of the WPP system and its performance improvements over existing alternatives.

Zhuque is faster than existing PMEM programming systems. It is between 4.7× and 10.14× faster than PMDK [50], Mnemosyne [63], Atlas [6] and Clobber-NVM [68] on STAMP applications. Zhuque is also more flexible than these systems: Since Zhuque is agnostic about the application's locking scheme, it can run the most recent version of memcached, while those systems cannot. As a result, our Zhuque-based persistent memcached is more than 7.5× faster than any similar system. We also demonstrate Zhuque's flexibility by running unmodified Python benchmarks with minimal performance loss.

The rest of this paper is organized as follows. Section 2 provides some background on PMEM and associated software systems. Section 3 describes fundamental limitations of prior art necessitating the WPP model. We discuss the WPP design and musl-based system implementation in Section 4 and Section 5, respectively. Section 6 showcases the performance of WPP. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Background

PMEM has introduced new possibilities for designing storage systems: programs can have byte-addressable access to terabytes of persistent data at near-DRAM latencies. However, utilizing PMEM in a both performant and programmer-friendly manner remains a challenging problem.

This section begins by describing our machine model, and then reviews existing general-purpose persistent memory programming models and their limitations to motivate WPP.

2.1 Machine Model

WPP is designed for a multi-core, cache-coherent machine equipped with PMEM (e.g. Intel DC Persistent Memory [28] or persistent CXL.mem devices [53]), and supporting *flush-on-fail* semantics, meaning that they provide a hardware guarantee that all in-flight and cached writes will reach PMEM in the event of an external power failure (as opposed to a fault in the machine or its onboard power supply). Such guarantees are provided by eADR-compliant platforms and NVDIMMs, and CXL platforms and devices supporting Global Persistent Flush (GPF). eADR and GPF are similar solutions targeting different device interfaces: the primary hardware requirement for both is that the platform must store sufficient energy to allow caches and internal device buffers to be drained to persistence after a power failure [1, 53].

On x86 systems, both eADR and GPF require system firmware to initiate and oversee the drain to persistence in response to a System Management Interrupt (SMI) [1, 13, 14]. Upon receiving this interrupt, the processor retires all in-flight instructions, drains all stores to the cache, and saves architectural state (register file etc.) to a designated per-core memory region before beginning execution of the SMI handler [16]. In both GPF and eADR, this handler first flushes the processor caches (and, for CXL, the caches of any CXL.cache device), and then proceeds to flush the buffers on the PMEM devices (NVDIMMs for eADR, CXL.mem devices for GPF) [1, 14].

2.2 Persistent Programming Models

Most existing persistent programming libraries rely on marking regions as *failure-atomic*, that is, all of the code region's effects will survive a failure or none will. Models differ in whether regions are explicitly marked (*transactional*) or inferred from locks (*FASE-based*). In addition, one work has proposed making the whole system persistent.

2.2.1 Transactional Libraries

Transactional PMEM libraries expect the programmer to explicitly mark failure atomic sections. For concurrency, these libraries either rely on off-the-shelf transactional memory systems or require the use of their own locks. For example,

NV-Heaps [9], Mnemosyne [63], and DudeTM [41] are built on existing transactional memory (TM) systems, and implement their failure-atomicity techniques (e.g. redo or undo logging) on top of those systems.

Meanwhile, transactional libraries that rely on locks generally expect transactions to acquire and release locks in a conservative, strong strict two-phase locking pattern [51, 64], that is: transactions acquire all locks at transaction begin, transactions release all locks at transaction commit, and locks are released in the order they are acquired. For example, PMDK [50], Pangolin [71] and Clobber-NVM [68] require applications to follow this lock pattern.

2.2.2 FASE-based libraries

Atlas [6] proposed the concept of failure-atomic sections (FASEs) as an alternative to transactions. A FASE is a failure-atomic operation which begins when a thread acquires its first lock and ends when it holds none — importantly, the final lock held may be different from the first lock. Because this locking scheme allows updates to be visible to other FASEs before a FASE commits, FASE-based libraries are required to track dependencies between threads, and roll back dependent FASEs in case of failure. Because FASEs are dynamically formed at runtime, user annotation is not required for existing lock-based code. NVThreads [26], JUSTDO [30], and iDO [42] follow this model.

2.2.3 Whole System Persistence

Instead of basing persistence on bounded sections of code, whole-system persistence (WSP) [47] focuses on the persistence of the entire system. WSP describes a system substantially similar to eADR and GPF, where an interrupt at power failure triggers the draining of volatile caches/buffers to persistence. This model requires no annotation and avoids the extra work done by transactional or FASE-based systems, but requires that large amounts of state be made persistent at the instant of failure, which until the advent of flush-on-fail systems was not possible.

3 Limitations of Prior Art

In this section, we argue that the existing programming models for persistent memory, namely transactional or FASE-based, necessitate an alternative path, especially when working with legacy code.

Fortunately, the emergence of persistent caches has enabled our efforts to develop a revitalize a model that does not fit either of these directions, namely, whole process persistence, in which all process state is preserved at a power failure.

3.1 Limitations of Transactions

The fact that many failure atomicity libraries leverage transactional memory is not surprising — transactions are commonly leveraged for durability within databases and file systems. When applied to (volatile) multi-threaded code, the transactional memory programming model simplifies concurrency by exporting to the programmer "single global lock" semantics, that is, the programmer should simply protect groups of accesses to shared data as "transactions," each of which are mutually exclusive. The transactional programming model is in theory appealing as programmers need not worry about data races on shared data, multiple locks, or parallel performance. To this transactional programming model, many failure atomicity libraries add persistence: transactions become both visible to other threads, and persistent, upon transaction completion.

In practice, however, despite decades of research and dedicated hardware support, (volatile) transactional memory has failed to become a common programming paradigm for general purpose multi-threaded code. Transactions generally mix poorly with both other synchronization methods (locks, barriers, condition variables, etc.) [4, 70] and IO [45, 52], tend to incur significant performance overhead when compared to fine grained locking [4, 19], and are incompatible with legacy multi-threaded code [52], whose locking discipline is rarely compatible without significant rewriting. Support for transactional memory in C++, for example, remains experimental [45]. There is no indication that persistent transactional memory systems will solve these problems, indeed, they appear to perpetuate them.

Generally, the transactional programming model is exported to the programmer using a scoped transaction, (e.g. `transaction{}`) and the library guarantees transactions will execute mutually exclusively (e.g. PMDK's C++ interface). However, for PMEM, transactional libraries may syntactically decouple mutual exclusion from failure atomicity due to language limitations (e.g. PMDK's C interface). In such an API, the library expects the programmer to first explicitly acquire the necessary locks to gain mutual exclusion before, subsequently, executing the transaction's failure atomic contents.

Despite this apparent separation, a transactional PMEM library's programming model imposes hard limits on the locking discipline - it expects that all transactional updates are mutually exclusive and isolated by the locking discipline. This restriction effectively forces the application to use a limited locking scheme such as single-global-lock or strong strict conservative two phase locking to protect any failure-atomic update. The programming model *explicitly disallows* releasing or acquiring a lock while executing a failure-atomic update.

For more complex locking schemes in which failure-atomic writes are visible to other threads before they are committed, the use of a FASE-based programming model is required,

and is often necessary for legacy programs as, in general, their existing synchronization fails to follow the restrictive transactional requirements.

3.2 Limitations of FASEs

Despite being, at first appearances, more compatible with legacy code, we argue the FASE-based model is also fundamentally flawed, or, at the very least, excessively permissive. The FASE model defines a failure-atomic code region as a "contiguous critical section," that is, it defines a failure-atomic code region as stretching from a thread's first lock acquire until the point where it holds no locks. While flexible with respect to locking scheme, this model requires tracking runtime dependencies between concurrently running failure-atomic code regions, which may not be isolated from each other. This permissiveness results in complicated and degenerate scenarios for recovery.

As a contribution of this work, we demonstrate that, for certain adversarial application patterns, any FASE-based system will either fail to recover or collapse into a degenerate case in which literally all program state must be logged for recovery, including volatile data never accessed within failure atomic regions — effectively, the FASE programming model requires whole process persistence for correctness.

Theorem 3.1 (FASE Limitation) *There exist applications for which, in order to consistently recover from a crash, a reasonably permissive FASE-based failure atomicity system requires all volatile program state be available at recovery.*

We prove this theorem by counterexample. This counterexample (Figure 1) can emerge naturally where two threads communicate via shared variables and one executes IO, a common pattern in event-based servers. In these servers, some threads handle the IO socket (thread 2 in example), some threads are application workers (thread 1), and they communicate via shared flags to manage outstanding requests. Detecting this pattern requires detailed reasoning about synchronization, and therefore prevents the blind use of FASEs on applications.

In the remainder of this section, we describe the counterexample and a brief sketch of our proof's reasoning. A full proof by contradiction, formal definitions, and additional discussion incorporating related work can be found in Appendix A.

Figure 1 gives our adversarial application that breaks FASE-based systems. In this example, two threads compute a fixed series of four values for nonvolatile variable x. Thread 1 computes the first value, Thread 2 the second and third, and Thread 1 the final, fourth value.

The two "tricks" of the code are that (1) the long FASE executed by thread 1 (lines 6 through 22) spans the entire example and (2) the third value of x, computed, but not assigned, outside of a FASE (line 39), is dependent on an access to a large volatile array Q.

```
1  lock_t lock0, lock1, lock2;
2  bool cond1 = false, cond2 = false;
3  int Q[] = rand(); // large random volatile array
4  nvm<int> x = 0; // x resides in nvm
```

```
5  void thread1{
6    lock0.lock();
7      x = (int s1=f1(x));
8
9      lock1.lock();
10       cond1 = true;
11     lock1.unlock();
12
13     bool w = true;
14     while(w){
15       lock2.lock();
16         if(cond2)
17           {w = false;}
18       lock2.unlock();
19     }
20
21     x = (int s4=f4(x));
22   lock0.unlock();
23 }
```

```
24 void thread2{
25   bool w = true;
26   while(w){
27     lock1.lock();
28       if(cond1){
29         w = false;
30         x =(int s2=f2(x));
31       }
32     lock1.unlock();
33   }
34
35   int in;
36   printf("x=%d", s2);
37   scanf("%d",&in);
38   /*****/
39   int s3 = f3(s2,in,Q);
40
41   lock2.lock();
42     x = s3;
43     cond2 = true;
44   lock2.unlock();
45 }
```

Figure 1: FASE counterexample

Recovery of this example presents an unsolvable problem. First, we note that Thread 1's long FASE, due to failure-atomicity semantics, forces recovery to recover either to the very beginning of the program or the very end. However, both options are impossible for a crash at line 38, just before x's third value is computed. At this point, thread 2 has already issued IO, so rolling back program state at recovery is inconsistent with the external world. However, rolling forward from this point requires the computation of the third value of x, which is dependent on an arbitrarily sized volatile array (Q). Since Q can be of any size, it can be replaced, without loss of generality, with any or all of the program's volatile state, effectively requiring whole process persistence.

Our proof requires failure atomicity systems to be "reasonably permissive," by which we mean that this counter example can be expressed as valid input for the system. Systems that restrict locking to two-phase-locking (e.g. [50,68]) or a single, semantic, global lock (i.e. transactional memory [63]) avoid this counterexample by prohibiting the locking pattern. Of course, by the same token, this restriction hampers their utility for legacy code, which rarely follows such a strict locking discipline.

The FASE programming model may be fixable by prohibiting situations like the counter-example. Simple (but undesirable) solutions include prohibiting all volatile accesses or all IO in the program. Alternatively, we could try to prohibit the precise counter-example problem by targeting the interplay

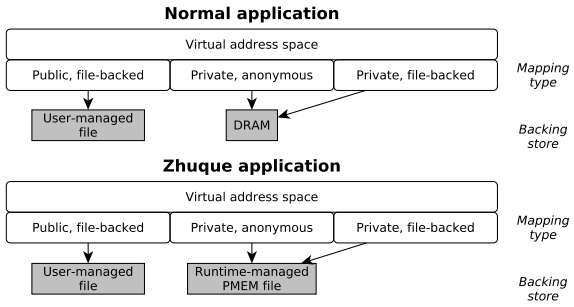


Figure 2: **Virtual memory in Zhuque.** The runtime modifies the backing store based on the mapping type, but the interface presented to the userspace application does not change.

between FASE dependencies, volatile accesses, and IO. One potential approach to achieve this involves a specification that disallows volatile accesses concurrently with a FASE execution. However, formally defining this specification is tricky, and formally verifying the proper use of FASEs is almost certainly undecidable through the halting problem. Notably, the requirement of a transactional locking scheme (e.g. strict, strong conservative 2PL) would also prevent the counterexample by explicitly disallowing its locking discipline.

To our knowledge, all existing FASE-based systems (e.g. [6, 26,30,42]) are "reasonably permissive" and would both accept this code as valid input and fail to recover correctly on it.

4 Design

Whole process persistence (WPP) is our answer to the limitations of transaction- and FASE-based programming models. In WPP, the in-memory state of an individual process is made persistent with, in simple cases, no modification to the application, primarily by interposing on the creation of virtual memory mappings (see Figure 2). WPP is designed for systems with flush-on-fail support, so we expect the contents of the process's PMEM-backed cache lines to survive a power failure. When the process is restarted after a power failure, it receives an OS signal, which it can ignore or handle with a signal handler. If no signal handler is installed, or if the installed signal handler does not exit the program, each thread continues execution at the point where it was interrupted by the failure.

There are several benefits to this model over transactions and FASEs. First and most importantly, WPP solves the problem described in Section 3 by discarding the concept of a failure-atomic section. The visible effects of an instruction on process state (that is, not including effects on OS state or peripherals) are guaranteed to survive a failure at least from the point at which they are visible to other threads. Second, restarting at the point of failure removes the need to "redo" or "undo" any writes at recovery, and with it the need to keep a persistent log and incur the cost of extra writes to PMEM. Third, no longer needing to define failure-atomic sections either reduces the programmer's burden directly, compared to manually-annotated failure-atomicity systems, or allows them to design concurrency schemes orthogonal to persistency without incurring overhead, unlike FASE-based systems.

There are two requirements that must be satisfied in order for an application to use WPP. The first is that its threading and virtual memory must be managed using a well-defined API for those purposes (i.e., on POSIX: `mmap()`, `pthread_create()`, etc). Any modern application targeting a POSIX system would have to go out of its way in order to violate this requirement.

The second is that applications must check error returns from system calls and other mechanisms that access non-process-private state, to detect failure-related errors beyond the process boundary, such as an application using a file on a filesystem that was not remounted after system restart. This requirement is more onerous than the first, but in our experience a wide range of applications can be correctly restarted without modification or special handling.

The principal challenge in implementing WPP is preserving process state across a power failure. Continuing execution after failure requires that the process's virtual address space, volatile architectural state, and relevant kernel-resident state (e.g., the file descriptor table) are a) persistent or b) can be resurrected along with the application.

The remainder of this section introduces Zhuque, our runtime implementing WPP, and describes how it makes process state persistent and restores that state after failure.

4.1 Overview

Zhuque provides WPP functionality by interposing on system calls which allocate resources (memory, file descriptors, threads), and by modifying the application startup process. In order to do this, we modified `libc`, which provides C bindings for system calls and implements the application startup process. Zhuque also requires small changes to the kernel to protect userspace context when failures occur in kernel mode (see Section 5.3 for details).

Interposing on system calls allows Zhuque to ensure that all application state which is normally volatile is instead stored in PMEM, as shown in Figure 2. It also allows Zhuque to track memory mappings and system calls so it can reconstruct the program's address space and re-create its kernel-resident state after a failure. Remaining volatile architectural state (e.g., the register file) is preserved by writing it to PMEM at failure.

When the application is resurrected after failure, Zhuque restores the application's address space, respawns its threads, and each thread reloads its architectural state. Execution resumes by calling the program's power failure signal handler, if it exists, and then resuming execution of each thread at the point interrupted by power failure.

4.2 Ensuring State Persistence

The first requirement that Zhuque must fulfill is ensuring that all state required for continuing correct execution of a program is preserved across power failures. This state can be divided into three categories based on its storage location: architectural state, memory state, and file state.

If a system supports flush-on-fail, it would be possible to modify its firmware to write per-thread architectural state (register file, floating-point configuration, etc.) to PMEM in response to power failure. However, we do not have the ability to modify that firmware, so we emulate it using userspace signals (see heading Power Failure in Section 5.1). We also save architectural state to PMEM on every kernel entry, in case a failure occurs in kernel mode (see Section 5.3).

File state is either inherently persistent, if the file was opened read-only or if changes have been written to disk, or is buffered awaiting being written to disk, in which case it is actually memory state and is handled as described below.

Automatically ensuring memory state is persistent is more complex, and is one of the main innovations of this design. Memory state itself can be divided into dynamic and static memory.

Dynamic memory Programs conjure dynamically allocated (heap) memory and thread stacks by calling anonymous `mmap()` (often via `malloc()`). Zhuque interposes on `mmap()` so that requests for anonymous memory return DAX-mapped persistent memory backed by a runtime-managed PMEM file, making heap and stack memory persistent.

Static memory Before an application binary is executed, the loader uses `mmap()` to create memory regions to hold code and static data (globals) from the application binary and linked dynamic libraries. Zhuque treats these regions differently based on whether they are un/zero-initialized, or initialized to non-zero values. The loader creates un/zero-initialized regions with anonymous `mmap()`, so they are treated as dynamic memory.

Initialized static memory, however, actually takes up space in the binary, and is loaded by mapping that region of the binary into memory as a private mapping. Thus, Zhuque transforms any writable, private mapping backed by a file to a writable, shared mapping that is backed by a PMEM file (see Figure 2), which is populated with the initialization values from the binary.

This mechanism also cleanly handles other outputs of dynamic loading, like relocations of position-independent code and cross-binary symbol resolutions, since they also are stored in writable, file-backed, private mappings.

4.3 Ensuring Correct Restoration

Having persisted the application's state, we also have to ensure it can be restored correctly. Recovery must restore the application address space, restore kernel-resident state, and restore architectural state.

Application address space All of the PMEM-backed memory mappings managed by Zhuque, as well as any other mappings the application created with `mmap()`. Zhuque stores the mapping table in a persistent memory file, and updates it to match any changes to the address space as they occur, so no action is required at failure to ensure this metadata is persistent.

At recovery, restoring the virtual memory map to its previous state must be done first, because all other state to be restored is stored in virtually mapped persistent memory. Restoration consists of re-mapping each virtual memory region with the correct backing store and access permissions. This restoration also replaces dynamic loading.

Kernel-resident state Any data required for continuing execution that resides outside the address space (and architectural state) of the process. The specific data varies depending on operating system and implementation decisions: for instance, Zhuque tracks the state of open Linux file descriptors in PMEM and restores them at restart using system calls. We discuss Zhuque's handling of kernel-resident state in Section 5.

Architectural state Any state stored in the processor itself and directly accessible from software. This state is per-thread, and since it includes the program counter and stack pointer registers, restoring it is equivalent to restarting execution of the thread (so it must be done last).

Zhuque manipulates the saved PC and stack so that the thread resumes as if it had just called the application-defined failure handler (if it exists), and then that handler returns to the point interrupted by execution when it executes a `RET` instruction. To avoid references to a thread which has not yet been recreated, threads wait to restart execution after they are created until all threads have been created.

5 Implementation

Zhuque is based on the musl implementation of the C standard library runtime [46], plus a minor modification to the Linux kernel (Section 5.3). Figure 3 depicts Zhuque's place in the runtime environment, and Figure 4 shows the changes to control flow at initialization and termination. This section describes the life cycle of a Zhuque process, describes how Zhuque handles the userspace-kernel boundary, and finally discusses some limitations of our prototype implementation.

5.1 Process Life Cycle

When Zhuque starts a process, it checks an environment variable for a path to a directory which holds or will hold the persistent state for that process. One file in the directory holds the process's global "process context", a memory map of a C

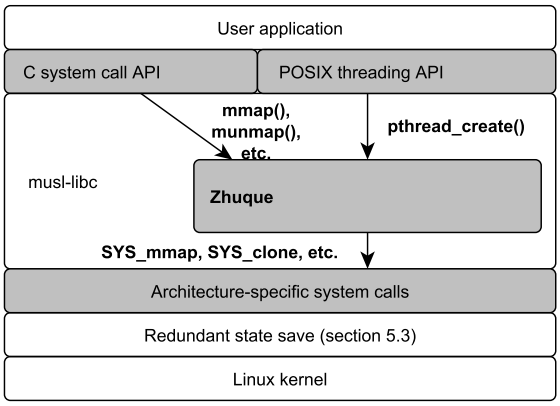


Figure 3: **Zhuque architecture.** User applications link to the C APIs provided by musl libc, and we modify the implementation of the APIs and the arguments passed to the underlying system calls. To protect against failures in kernel mode, we save userspace context to PMEM on entry to the kernel.

structure. The directory also holds all other persistent memory files allocated during the process's life.

Zhuque takes control of the process after the dynamic loader loads its own metadata using information provided by the kernel ("loader bootstrapping"). If the context file is present, then Zhuque takes steps to *restart* the process. If the context file is missing, but the environment variable is set, then it is a newly created Zhuque process (i.e., a *clean start*).

Clean start In the clean start case, Zhuque creates and initializes the process context file. Then, it records the locations of the dynamic loader and the main binary in the mapping table and remaps their static memory sections to memory backed by a persistent file. This retroactive process is necessary because our userspace runtime cannot interpose on mappings created by the kernel.

Next, control returns to the loader and it loads the application's dynamically-linked dependencies. Our code intercepts the loader's calls to `mmap()` and `mprotect()` during this process in order to record the mapping metadata and transform any writable, private mappings into persistent memory regions.

After loading is complete, control returns to Zhuque just before `main()` executes. Zhuque copies `main()`'s arguments into PMEM and runs it in a new thread with a persistent stack.

Power failure To save volatile architectural state (e.g. the register file) to PMEM at failure, we propose repurposing existing functionality. NVDIMM eADR and CXL GPF both rely on a System Management Interrupt (SMI) to implement the flush-on-fail process on x86 systems (see Section 2.1). SMI handling saves volatile architectural state to a designated per-core region (the *SMRAM*) before beginning execution of the handler, and x86 allows the SMRAM to be PMEM-backed [16]. However, the location of the SMRAM is controlled by system firmware. Unfortunately, updates to firmware must be signed by the manufacturer — the firmware uses encryption to prevent modification by the end user [15], so we were unable to make this change for our prototype.

Instead, to test Zhuque's application support, we emulate the SMI's state save using userspace signals. If SIGPWR is delivered while the process is executing, the volatile thread receives it and sends a second signal to each thread. When the kernel interrupts a thread to run the signal handler, it first pushes the register file and other state needed to resume execution onto the persistent thread stack. The handler body saves the current stack pointer and some context not saved by handler entry in PMEM, and then exits the thread directly, preserving the contents of the stack. Thus, at recovery, we have access to a persistent memory region containing a snapshot of volatile architectural state at failure, as if it had been saved by an SMI.

Restart after failure On restart, the runtime opens the context file, re-creates PMEM mappings, re-opens file descriptors, and finally re-maps file-backed memory. If a file descriptor was closed after being used to create a mapping, it is temporarily re-opened while the mapping is restored.

After the virtual memory map and file set are re-established, Zhuque restarts the execution of each thread from the point of failure. Zhuque does this by starting each thread with the same start routine, and the same initial stack pointer, so that the bottom frames of the stack are overwritten with new frames of the same size, and the contents of application frames are preserved. From this entry routine, we use assembly to restore architectural state, including setting the stack pointer and PC to the addresses saved at failure. Execution resumes within the runtime's failure handler, which calls the user-defined failure handler if present. If there is no user-defined handler, or the handler does not exit the program, execution continues at the point interrupted by the signal at failure.

5.2 Kernel-resident State

In order to preserve correctness in a userspace-only implementation, our runtime tracks and restores two pieces of kernel state tied to the process: the file descriptor set and the thread set.

To track the **thread set**, our runtime interposes on calls to `pthread_create()`, wrapping the passed thread entry point and arguments in our own entry point function (which itself is wrapped in the musl entry point function). It also saves both the Linux and pthreads identifiers for the thread; the Linux ID is used at failure to signal each thread individually with `tgkill()`, while the pthreads ID is the address of the thread metadata, and is used at restoration to continue execution at the point of failure. The Linux ID of a thread changes when the process is restarted, while the pthreads ID does not, because we restart threads at recovery with a modified version of `pthread_create()` which uses an existing thread metadata object rather than creating a new one.

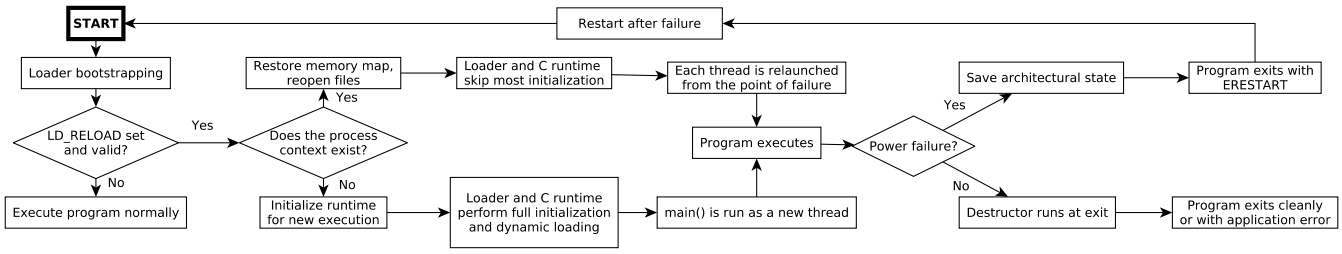


Figure 4: **Zhuque runtime control flow.** Zhuque modifies runtime startup and termination; application code is not modified.

To track the **file descriptor set**, Zhuque interposes on calls which assign (e.g. `open()`, `socket()`), modify (e.g. `fcntl()`, `bind()`), or release (i.e. `close()`) file descriptors and replays them at restart, using `dup()` to patch any discrepancy in assigned descriptors. This approach is sufficient for sockets with stateless protocols, `epoll` file descriptors, and simple file accesses.

However, pipes and files require special handling: for regular files, we ensure that they will not be deleted between failure and restoration by creating separate hardlinks to the files and using those to reference the file, deleting them when the program exits cleanly; we also open them without kernel buffering (i.e. with O_SYNC) due to the limits of a userspace implementation. And for pipes, we use the `splice()` family of system calls to save and restore unconsumed contents at failure/restoration. Support for restoring network sockets is best-effort, and a more systematic approach to network support under WPP is an interesting future extension of this work.

5.3 Failures in kernel mode

When an application makes a system call, or is suspended by an interrupt, the kernel will save the application's volatile architectural state on the suspended thread's kernel stack and restore it when application execution resumes. In our implementation the kernel stack is volatile, so we must save this state in persistent memory to allow recovery if power failure interrupts the kernel-mode operation.

To enable this, we added a `prctl()` operation to designate a page as a redundant state save area. We added code on all entries from user- to kernel-mode which checks whether such a page has been provided, and if so saves the state there as well as to the kernel stack. Accesses to the page must not fault: since a page fault is itself an interrupt, a fault in interrupt entry deadlocks the kernel. We found that there is no way (in our test kernel version) to reliably prevent access to a filesystem DAX page from faulting, so we use device DAX to provide the save memory.

5.4 Limitations

There are two notable limitations of our implementation, neither of which is fundamental to the design.

Multi-process applications are not supported. Zhuque currently has no support for persisting multiple processes in the same process tree; if an application under our runtime forks a new process while leaving the `LD_RELOAD` environment variable unchanged, the child process will crash when it attempts to use the same context object as its parent. By the same token, we make no attempt to preserve OS process IDs across failures, so applications that save and retrieve their PID after failure may find it invalid. Our runtime supports unrestricted concurrency schemes, so we believe it would be possible to extend it to support multi-process operation by interposing on the creation and termination of processes, similar to our approach to threads.

Some ASLR is not supported. Address space layout randomization (ASLR) is a security technique which randomizes the address of virtual memory mappings. Random addresses returned by `mmap()` to userspace are not an obstacle, since the randomization only occurs once under Zhuque. However, Zhuque cannot prevent the kernel from mapping libc and the application binary at a random location at restart, which means that their static memory cannot be recreated at the same locations when ASLR is enabled. It would be possible to move those mappings after they are created, but since it does not otherwise affect correctness or performance, and it would add significant complexity to the startup process, we chose not to implement this feature.

6 Evaluation

In this section, we evaluate Zhuque's performance to provide answers to the following questions:

- How much performance improvement does Zhuque provide for persistent applications compared to existing libraries?
- How much performance overhead does Zhuque incur compared to native, volatile execution?
- What benefits does Zhuque provide by enabling zero-

effort persistence?

Our experimental workloads include a set of microbenchmarks, a set of Python benchmarks, and three recoverable applications.

6.1 Evaluation Setup

We compare against native application performance based on musl and the performance of four popular PMEM libraries.

Musl [46] stands for the default (volatile) implementation based on musl libc.

PMDK [50] is Intel's failure atomicity library. It uses hybrid undo-redo log for both failure-atomicity [27] and memory allocation [55]. **Atlas** [6] also uses an undo logging-based mechanism for failure-atomicity. It can automatically infer failure-atomic region boundaries by analyzing lock behaviors in application code. **Clobber-NVM** [68] is a state-of-the-art PMEM library. It records clobber_log and v_log during runtime, and recovers an application by re-executing any interrupted transactions. **Mnemosyne** [63] is a redo-log based system. Instead of relying on locks in user applications, Mnemosyne uses the C++ transactional memory model to parallelize code.

To measure their performance with flush-on-fail semantics, we removed the flushes and fences from all three comparison libraries. These *flush-on-fail (FoF)* versions are used for all experiments described below, unless otherwise noted.

We run the benchmarks on a platform with one 20-core Intel Xeon Gold 6230 processor, running at 2.1 GHz. The platform has a total of 96 GB of DRAM and 768 GB (6 ×128 GB) of Intel Optane DC Persistent Memory directly attached to the processor [28]. We configured our test machine such that Optane DCPMM is in 100% App Direct mode [2]. In this mode, software has direct, byte-granularity access to the Optane DCPMM. Zhuque uses DAX-mapped [40] `Ext4` files for all PMEM allocations except the kernel state save area, which uses device DAX for the reasons described in Section 5.3. Unless otherwise noted, all Zhuque experiments are run on the modified kernel with the state save area activated, while all comparison software is run on an unmodified kernel of the same version (Ubuntu kernel 4.15.0-169). Each data point reported is the average over five runs.

We observed variations in memory usage across different benchmarks, ranging from 20 MB to 1 GB. For all but two of the Python benchmarks, the memory usage exceeds the Last Level Cache (LLC) capacity of 30.25 MB.

As discussed in Section 5.1, we were unable to modify platform firmware to direct the SMI state save to PMEM, because firmware updates must be signed by the manufacturer [15]. However, since it only affects events at failure, we expect that making this change would produce no measurable changes in the steady-state performance results we report below.

We verified that, with Zhuque enabled, all benchmark applications restarted and ran to completion correctly after randomly-timed simulated power failures.

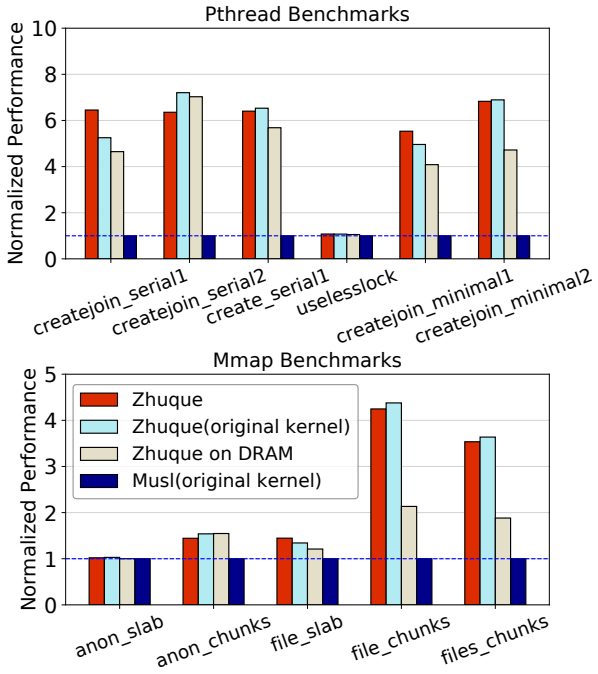


Figure 5: **Measuring the overhead of Zhuque on basic operations.** Performance values are normalized to the Musl(original kernel) value.

6.2 Microbenchmarks

In our first experiment, we compare Zhuque's performance with the default (volatile) musl libc on a set of microbenchmarks from the libc-bench [37] benchmark suite which tests the operations modified by Zhuque: thread creation and virtual memory mapping. We implemented the latter within the libc-bench test harness. Descriptions of the `pthread_create()` benchmarks can be found in the libc-bench documentation [37]. The `mmap()` benchmarks are described below.

anon_slab creates a 16 MB anonymous mapping, writes to every page, and then removes the mapping. **anon_chunks** creates, writes to every page, and then removes 128 128 kB anonymous mappings. **file_slab** creates a 16 MB file-backed mapping, writes to every page, and then removes the mapping. **file_chunks** creates, writes to every page, and then removes 128 128 kB mappings backed by a 16 MB file.
files_chunks creates, writes to every page, and then removes 128 128 kB mappings each backed by a separate 128 kB file.

We ran these benchmarks on four implementations: **Zhuque** is Zhuque using DAX-mapped PMEM as the backing store, running on modified kernel. **Zhuque(original kernel)** is Zhuque without kernel modification. **Zhuque on DRAM** uses non-PMEM files (loads and stores access the DRAM page cache) as a backing store, but also saves kernel

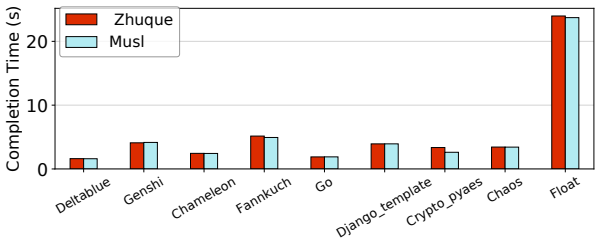


Figure 6: **Measuring the overhead of different Python benchmarks**: Zhuque matches native performance on some benchmarks

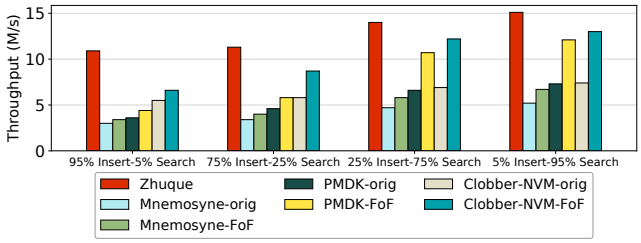


Figure 7: **Porting Existing PMEM Libraries to Flush-on-Fail Machines Provides up to 1.76× Improvement**

states. **Musl(original kernel)** is unmodified musl libc with unmodified kernel. We report the results in Figure 5.

We observe that Zhuque introduces significant overheads to modified operations, especially `mmap()` and `munmap()` – the bottleneck for all of the poorly-performing microbenchmarks is the allocation of new anonymous memory. The overhead is per-operation, and does not depend on the size of the mapping. This is not surprising: our modified versions still perform the original operations, and are also required to modify userspace data structures maintained by Zhuque in persistent memory, often including a search whose time is linear in the number of created mappings. As demonstrated by the results below, these overheads do not translate to a significant slowdown on macrobenchmarks, because modifications of the virtual memory map are rarely on an application's critical path. In addition, these overheads are a result of implementation choices, not the fundamental design. We anticipate they would decrease substantially in a kernel-based implementation of WPP.

6.3 Python Benchmarks

In this experiment, we ran nine Python benchmarks on the musl and Zhuque configurations using the CPython interpreter. It demonstrates that Zhuque can run a wide range of unmodified Python applications. We chose the first nine benchmarks (out of 42), in alphabetical order, from version 1.0.0 of the Pyperformance benchmark [21] suite. Descriptions of the benchmarks can be found in the Pyperformance documentation [21]. We ran them in our own benchmark framework, but did not modify the benchmarks themselves. We also added dynamic thread stack support, not present in vanilla musl, in order to support the large stack sizes required by the interpreter. We report the results in Figure 6.

Most of the Python benchmarks perform competitively with the musl versions. Those that perform worse generally incur overhead from frequent random-access reads and writes to data structures too large to fit in the cache, causing thrashing and exposing the higher access latency of persistent memory compared to DRAM.

6.4 Memcached

Memcached [44] is a widely-deployed key-value store. Early versions (1.2.*) have been ported to Mnemosyne, PMDK, and Clobber-NVM. We ran memcached-1.2.5 on Zhuque unmodified.

We evaluate memcached performance with four types of workloads: insertion-intensive (95% insertion / 5% search), insertion-mostly (75% insertion / 25% search), search-mostly (25% insertion / 75% search), and search-intensive (5% insertion / 95% search). We use memslap [39] to generate a stream of uniformly distributed memcached requests with 16-byte keys and 64-byte values. As shown in Figure 7, Mnemosyne, PMDK and Clobber-NVM can perform up to 1.76× faster with flushes and fences removed. This result indicates that the flush-on-fail semantics can benefit existing PMEM libraries.

Figure 8 presents the results of these experiments, using Musl-based memcached-1.2.5 performance on DRAM as baseline. We find Zhuque can always provide nearly 80% of musl memcached performance. Across different thread configurations, Zhuque provides up to 3.58× Mnemnosyne's throughput, 1.81× of PMDK's throughput and 1.71× of Clobber-NVM's throughput.

Poor scalability is a well-known problem with early versions of memcached [20, 42]. Memcached went through a rewrite of the synchronization framework to use fine-grained locking across seven years of development and over thirty versions [31]. Many current (transactional) PMEM libraries have strict requirements for applications' concurrency schemes. These requirements make converting recent versions of memcached to run on PMEM a complicated and difficult process. Zhuque places no restrictions on the locking scheme, so the newest version (1.6.17) can run unmodified on Zhuque. By simply running the newest version on Zhuque, we can provide more than 7.5× performance of the best performant older version of persistent memcached on the same workload, with the same thread count.

6.5 Vacation and Yada

Furthermore, we evaluated the performance of the Vacation and Yada applications from the STAMP benchmark suite [8],

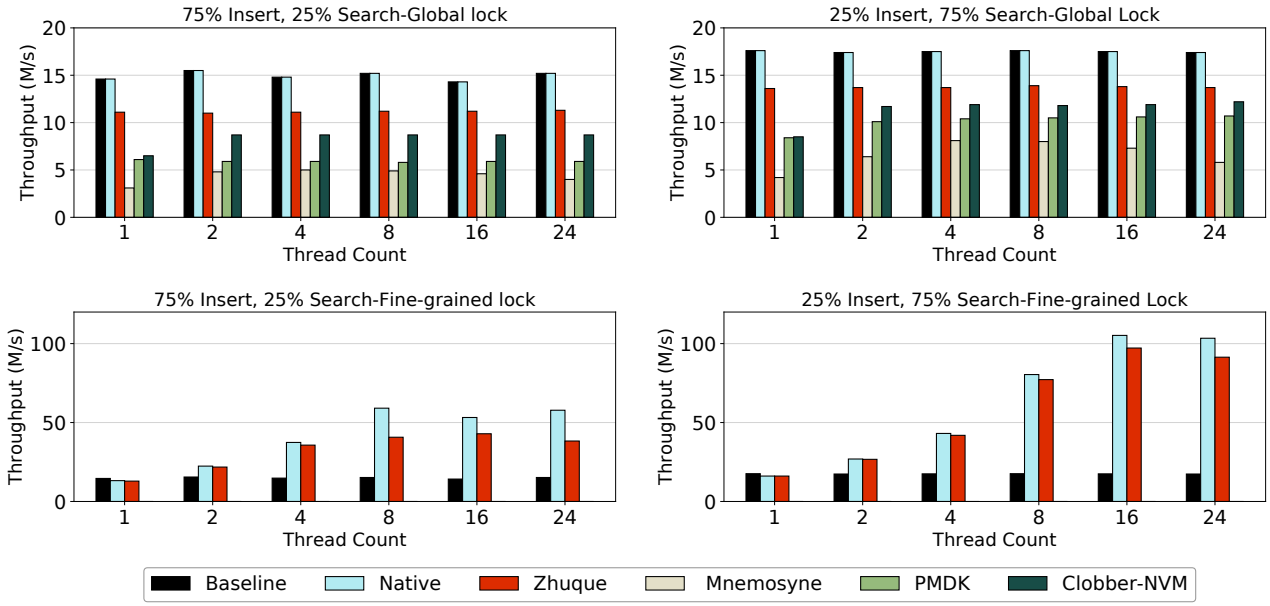


Figure 8: **Zhuque Enables Newest Version of Memcached to Run on PMEM, Provides Significantly Better Performance**

each targeting common PMEM applications such as, respectively, KV-stores and graph workloads. Prior works [25,63,68] provide readily available implementations of these applications built on top of existing PMEM libraries, with the exception of Yada - Mnemosyne.

We compare Zhuque with Musl, Mnemosyne, PMDK, Atlas and Clobber-NVM Vacation performance. Vacation is a travel reservation system, consisting of tables updated concurrently using transactions that span multiple tables.

Prior implementations [25, 63, 68] persist the tables in PMEM, and leave the client side in volatile memory. Zhuque persists the entire vacation application, including both the server tables and the client threads.

Figure 9 shows that Zhuque performs 10.8×, 4.8×, 3.7× and 3.6× faster on Vacation than Mnemosyne, PMDK, Atlas and Clobber-NVM, respectively. The performance gain comes from fewer logging writes and more efficient memory management: Zhuque uses vanilla `malloc()`, while the other libraries use either PMDK's transactional allocator (PMDK, Clobber-NVM) or a hand-written allocator (Mnemosyne, Atlas). The memory management efficiency problem is more prominent on Yada, which implements Ruppert's algorithm for Delaunay mesh refinement [54].

7 Related Work

PMEM Software & Hardware. In the past decade, researchers have designed many highly-optimized PMEM data structures [7, 10, 22, 48, 62, 69]. They are designed to reduce the cost of persistent updates while ensuring failure-atomicity. Because they are carefully designed by experts to cope with PMEM characteristics, they usually provide good performance. However, using and developing these data structures takes significant programming effort.

Because PMEM's bandwidth is significantly higher than traditional secondary storage, PMEM file systems [11, 33, 65, 67] aim to expose raw PMEM performance as much as possible. It is easy for existing applications to use files on these PMEM file systems, but they are not designed to solve the same problem that Zhuque targets (failure-resilience of the application's in-memory data).

The architecture community has also sought better hardware support for PMEM, often by allowing more permissive (and thus performant) store ordering at the memory controller [3,18,24,32,34–36,59]. Many of these systems demonstrate dramatic performance gains in simulation, but none that we are aware of are available in production.

General-purpose PMEM libraries. General-purpose PMEM libraries aim to ensure failure-atomicity for applications which directly access PMEM, with low overhead and minimal code changes.

Traditional undo-log systems [6, 9, 50] and redo-log systems [23, 63] write to a log alongside every visible update, at least doubling each write. On non-flush-on-fail PMEM machines, undo-logging usually requires expensive memory fences at the end of each log write, while redo-logging needs to redirect loads even if the machine supports flush-on-fail.

To avoid expensive synchronization between threads, some undo/redo systems buffer writes in "shadow" copies of PMEM data [5,17,41,43,66], at least doubling the amount of PMEM required by the application. These systems still incur the write amplification and read redirection costs of conventional log-

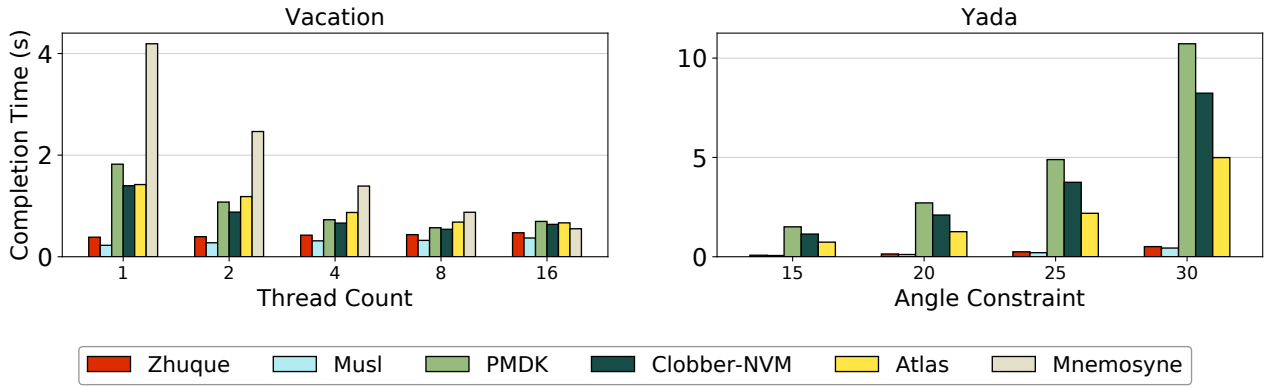


Figure 9: **Vacation and Yada performance on different PMEM libraries and Zhuque**

ging, and with flush-on-fail semantics the synchronization they are designed to reduce is no longer required at all. Compared to these systems, Zhuque does not amplify or redirect memory accesses, nor increase the size of the working set.

All these systems rely on either lock-inferred failure atomic sections (FASEs) [6, 26, 30, 42, 66], classical transactions [5, 41, 43, 63], or programmer delineated transaction boundaries with a restricted lock scheme [23, 50, 68] to identify failure-atomic operations. In contrast, Zhuque is not concerned with synchronization: it uses the same concurrency model as the original application.

JUSTDO [30], iDO [42], and Clobber-NVM [68] recover by resuming execution of the interrupted failure-atomic section or re-executing interrupted transactions. These systems are similar to Zhuque in that they also resume execution at the point of failure, but they all restrict concurrency and require manual annotation of atomic sections.

Single Level Stores. WPP transparently makes processes persistent by providing continuous checkpointing, durability guarantees for external observers of application IO (known as *external synchrony* [49]), and POSIX compatibility. Single level stores (SLS) [12, 38, 56–58, 60, 61] also provide persistent address spaces to applications, but they suffer from high overhead, rarely support external synchrony, and are often hard to use due to custom APIs [57, 60].

The high overhead arises from checkpointing and, for some systems, the enforcement of external synchrony. Checkpointing overhead is high because traditional storage devices are far slower than DRAM, and SLSes usually amplify writes by tracking memory modification at page granularity. To achieve external synchrony, SLS systems must delay IO until data is safely persisted. If checkpoints are too frequent, the communication delay can cause high overheads.

Because of the performance penalty of providing external synchrony, most SLSes choose not to enforce it. The state-of-the-art SLS system Aurora [61] points out the value of external synchrony, but does not support it. Zhuque shows that flush-on-fail semantics allow continuous checkpointing, making external synchrony feasible and performant.

8 Conclusion

This paper has described Whole Process Persistence (WPP), a programming model that treats power failure as a recoverable exception. Zhuque, implementing WPP, transparently makes applications failure-resilient by interposing on the POSIX-specified APIs. Zhuque ensures process state survives power failures, and allows for resumption of execution.

Compared with existing solutions, Zhuque greatly simplifies the programming model. Our evaluation shows that Zhuque significantly improves performance compared to state-of-the-art, and tends to match original volatile application performance on certain workloads.

Acknowledgments

This work was supported in part by the NSF/Intel Foundations of Microarchitecture Program awards 2011213 and 2011212. We would like to thank our shepherd Gaël Thomas and the anonymous reviewers for their insightful feedback.

References

[1] eadr characteristics at failure. https://groups.google.com/g/pmem/c/K35X70fzAMw/m/5qEhhzb8AAAJ, 2021.

[2] Alper Ilkbahar. Intel Optane DC Persistent Memory Operating Modes Explained, 2018.

[3] Miao Cai, Chance C Coats, and Jian Huang. Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596. IEEE, 2020.

[4] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Sid-

dhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11):40–46, November 2008.

[5] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 368–377, 2018.

[6] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452. ACM, 2014.

[7] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[8] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[10] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 441–454. Association for Computing Machinery, 2019.

[11] 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 ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[12] George Copeland, Michael Franklin, and Gerhard Weikum. Uniform object management. In *International Conference on Extending Database Technology*, pages 253–268. Springer, 1990.

[13] Intel Corporation. Asynchronous event handling. In *CXL Type 3 Memory Device Software Guide*, page 65. June 2021. Revision 1.0.

[14] Intel Corporation. Gpf sequence. In *CXL Type 3 Memory Device Software Guide*, page 121. June 2021. Revision 1.0.

[15] Intel Corporation. Microcode update facilities: Update signature and verification. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 10.11, pages 10–36–10–37. March 2023. Order No. 325462-079US.

[16] Intel Corporation. System management mode: Smram. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 32.4, pages 32–4–32–9. March 2023. Order No. 325462-079US.

[17] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 271–282. Association for Computing Machinery, 2018.

[18] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. Lazy release persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1173–1186. Association for Computing Machinery, 2020.

[19] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 188–197. Association for Computing Machinery, 2014.

[20] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2), February 2015.

[21] Python Software Foundation. The python performacne benchmark suite, 2021.

[22] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 28–40. Association for Computing Machinery, 2018.

[23] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[24] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665. IEEE, 2020.

[25] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, AS-

PLOS '20, pages 775–788. Association for Computing Machinery, 2020.

[26] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482. Association for Computing Machinery, 2017.

[27] Intel Corporation. Pmdk issues: introduce hybrid transactions, 2017.

[28] Intel Corporation. Intel Optane DC Persistent Memory, 2019.

[29] Intel Corporation. eADR: New Opportunities for Persistent Memory Applications, 2021.

[30] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. ACM.

[31] Joseph Izraelevitz, Lingxiang Xiang, and Michael L Scott. Performance improvement via always-abort htm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 79–90. IEEE, 2017.

[32] Jungi Jeong and Changhee Jung. Pmem-spec: persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–529, 2021.

[33] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[34] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411. Association for Computing Machinery, 2016.

[35] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[36] Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. Vorpal: Vector clock ordering for large persistent memory systems. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 435–444. Association for Computing Machinery, 2019.

[37] Eta Labs. libc-bench, 2021.

[38] Charles R Landau. The checkpoint mechanism in keykos. In *[1992] Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, 1992.

[39] libMemcached.org. libMemcached, 2011.

[40] Linux Kernel Organization. Direct Access for Files, 2020.

[41] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343. Association for Computing Machinery, 2017.

[42] 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. IEEE, 2018.

[43] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512. Association for Computing Machinery, 2017.

[44] Memcached. http://memcached.org/.

[45] Transactional memory study group (SG5). Technical specification for c++ extensions for transactional memory iso/iec ts 19841:2015, 2015.

[46] musl libc, 2021. https://musl.libc.org/.

[47] Dushyanth Narayanan and Orion Hodson. Whole-system persistence with non-volatile memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[48] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[49] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–26, 2008.

[50] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[51] Yoav Raz. The principle of commitment ordering, or

guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 292–312. Morgan Kaufmann Publishers Inc., 1992.

[52] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using gcc and memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 399–412. Association for Computing Machinery, 2014.

[53] Andy Rudoff, Chet Douglas, and Tiffany Kasanicky. Persistent memory in cxl. In *Proceedings of the 2021 SNIA Persistent Memory + Computational Storage Summit*, April 2021.

[54] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 1995.

[55] Steve Scargall. *PMDK Internals: Important Algorithms and Data Structures*, pages 313–331. Apress, 2020.

[56] Jonathan S Shapiro and Jonathan Adams. Design evolution of the eros single-level store. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2002.

[57] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[58] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

[59] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 175–186. Association for Computing Machinery, 2017.

[60] Frank G Soltis. *Fortress Rochester: The Inside Story of the IBM iSeries*. System iNetwork, 2001.

[61] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora operating system: revisiting the single level store. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 136–143, 2021.

[62] 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 Stroage Technologies*, FAST'11, page 5. USENIX Association, 2011.

[63] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.

[64] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.

[65] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[66] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. Pmthreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 623–637. Association for Computing Machinery, 2020.

[67] Jian Xu and Steven Swanson. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 323–338, 2016.

[68] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: Log less, re-execute more. In *To appear in the Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[69] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181. USENIX Association, 2015.

[70] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 265–274. Association for Computing Machinery, 2008.

[71] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 897–912, 2019.

A Proof

In this appendix, we provide a proof of theorem 3.1:

Theorem 3.1 (FASE Limitation) *There exist applications for which, in order to consistently recover from a crash, a*

reasonably permissive FASE-based failure atomicity system requires all volatile program state be available at recovery.

A.1 Definitions

We begin by defining terms. By *application* we mean a multi-threaded program, executed as a *process*. The process's *internal state* consists of all its data, including heap, globals, and stack. Some memory locations are designated *nonvolatile*, their contents (the *nonvolatile state*) survive a power outage; the remainder are *volatile*, and their contents (the *volatile state*) are lost. The process may perform IO operations — we term the set of IO operations performed by an executing process its *external state*. The process, being multi-threaded, contains code regions that execute while a lock is held, these are termed *critical sections*.

If power is lost during process execution, its volatile state is lost. The purpose of a *failure atomicity system* is to provide *consistent recovery* from a power outage. For consistent recovery, the system selects a point in execution, termed the *recovery point*. The recovery point is *consistent* with the external state; process execution from initialization through the recovery point would generate the observed IO. For failure atomicity, the recovery point also lies outside all critical sections. Consistent recovery of a process consists of selecting a valid recovery point and restoring the persistent state's contents to its values as of this point. If the power failure interrupts a critical section, consistent recovery will involve, for failure atomicity, chosing a recovery point outside the critical section and undoing or redoing changes made within the section.

We assume a powerful failure atomicity system which is free, during pre-crash execution, to intercept the process at any point and log data in nonvolatile memory. After a crash, the system has access both to these logs and the process's nonvolatile state — its task is to ensure that the nonvolatile state is restored to a recovery point; consistent with IO operations and outside any critical section. The failure atomicity system must be *reasonably permissive* with respect to its programming model — we require the system's programming model to support our adversarial example. To all our knowledge, all existing FASE-based systems are "reasonably permissive".

Figure 1 gives our counterexample. The "trick" is that the long FASE executed by thread 1 (lines 6 through 22) is dependent on non-FASE code executed by thread 2 that contains both IO and accesses to large volatile data (lines 36 through 39).

A.2 Proof Sketch

We prove Theorem 3.1 by contradiction. We consider a process executing the code sample in Figure 1 and suffering a power failure on line 38. Suppose, for contradiction, there exists a FASE system for which, given this situation, could restore the program's nonvolatile state to a recovery point consistent with the external state and outside any critical section. As thread 1, by construction, executes a critical section (FASE) for its duration, our recovery point for thread 1 must lie at line 6 or line 22 — all other points violate failure atomicity. We consider both options.

Suppose the recovery point lies at line 6 (i.e. recover `x` to 0), it is inconsistent with the external process state due to the IO executed before the failure on lines 36 and lines 37, which indicate that thread 2 (and therefore thread 1) have progressed beyond this recovery point, leading to a contradiction.

Suppose the recovery point lies at line 22 (i.e. recover `x` to `s4`). First we note that the value `s4` has a true dependence (read-after-write) on `s3`, and `s3` has a true dependence on both the inputed seed `in` and large volatile array `Q`. Since `s3` cannot be computed before `in` is known, `s3` must be computed after the `scanf` on line 37 is executed. Since the failure can interrupt the computation of `s3` after the `scanf`, all inputs to `f3` must be preserved in nonvolatile storage for recovery. However, since `Q` is an arbitrarily sized volatile array, Q can be of any size and can be replaced, without loss of generality, with any or all of the program's volatile state, requiring the failure atomicity system to preserve all volatile process state and leading to a contradiction.

B Artifact Appendix

B.1 Abstract

This appendix describes the artifact submitted with this paper. The artifact contains files to build a Docker image with Zhuque installed as the system `libc`, and containing all benchmarks and comparison PMEM systems evaluated in Section 6. It also contains our patch against the Linux kernel necessary for correct resumption, described in Section 5.3.

B.2 Scope

The artifact allows verification of the following claims:

- All performance results from Section 6, for both Zhuque and comparison systems.
- Zhuque can successfully restart programs after an simulated asynchronous failure, as described in Section 5.1.
- The kernel modification correctly saves userspace architectural state to the redundant state save area, as described in Section 5.3.

The artifact does not verify the following claims:

- The kernel modification is sufficient to protect against a failure in kernel mode (we cannot simulate this type of failure).
- The formal claims made in Section 3 about FASE-based systems.

B.3 Contents

The artifact is organized into these key directories (see README for detailed listing):

- `musl-src`: Source code of Zhuque-musl.
- `musl-src/src/psys`: Zhuque core implementation.
- `clobber-pmdk`: Source code for comparison PMEM systems and their versions of application benchmarks.
- `apps`: Zhuque/native implementations of application benchmarks.
- `pigframe`: Materials to build and test our kernel modification.

B.4 Hosting

This artifact is hosted in a Github repository at https://github.com/georgehodgkins/Zhuque_artifact. The commit ID for the current version is `ffc033972bb36adc23b7a4b8c8b2cc6d736bff53`. See README for build instructions.

B.5 Requirements

The only software required for the artifact is Docker on a Linux kernel; the build process bootstraps all other dependencies. Zhuque and most comparison applications can be run on a system without PMEM, but only a system with PMEM can fully reproduce the reported results. Zhuque was mostly developed against a rather old kernel version (4.15.18), and we have sometimes observed unexpected behavior when running on newer kernels.

We built and tested the artifact on the evaluation machine described in Section 6. Our kernel modification targets the Ubuntu kernel fork at version 4.15.0-169. The Docker image is based on Alpine Linux 3.14.

ENVPIPE: Performance-preserving DNN Training Framework for Saving Energy

Sangjin Choi *KAIST*
Inhoe Koo *KAIST*
Jeongseob Ahn *Ajou University*
Myeongjae Jeon *UNIST*
Youngjin Kwon *KAIST*

Energy saving is a crucial mission for data center providers. Among many services, DNN training and inference are significant contributors to energy consumption. This work focuses on saving energy in multi-GPU DNN training. Typically, energy savings come at the cost of some degree of performance degradation. However, determining the acceptable level of performance degradation for a long-running training job can be difficult.

This work proposes ENVPIPE, an energy-saving DNN training framework. ENVPIPE aims to maximize energy saving while maintaining negligible performance slowdown. ENVPIPE takes advantage of slack time created by bubbles in pipeline parallelism. It schedules pipeline units to place bubbles after pipeline units as frequently as possible and then stretches the execution time of pipeline units by lowering the SM frequency. During this process, ENVPIPE does not modify hyperparameters or pipeline dependencies, preserving the original accuracy of the training task. It selectively lowers the SM frequency of pipeline units to avoid performance degradation. We implement ENVPIPE as a library using PyTorch and demonstrate that it can save up to 25.2% energy in single-node training with 4 GPUs and 28.4% in multi-node training with 16 GPUs, while keeping performance degradation to less than 1%.

1 Introduction

Reducing carbon footprint is a worldwide mission. Experts estimate data centers take up 3% of the global carbon emission, which is roughly equal to the worldwide airline industry [2]. To mitigate carbon emissions, data center providers should actively explore energy-saving strategies for their operations. One significant area to address is the energy consumption associated with machine learning (ML) workloads, which constitutes a significant portion of overall energy usage. According to recent work, Google constantly spends 15% of its total energy running ML workloads for the past three years [22]. This study primarily focuses on energy saving in the context of multi-GPU deep neural network (DNN) training, which is a prevalent method employed in modern ML workloads.

There have been several approaches to save energy of GPU workloads. Common methods use GPU *Dynamic Voltage and Frequency Scaling* (DVFS), which seeks to identify the optimal frequency for the Streaming Multiprocessor (SM) clock or memory clock by balancing the tradeoff between performance and energy consumption [3, 7, 9, 12, 17, 27–29]. Recently, Zeus [29], considers the batch size and power limit to navigate the tradeoff between performance and energy consumption. It automatically finds the optimal configuration in recurring DNN training jobs based on the user-provided energy-efficiency importance. Although effective, these approaches leave several limitations.

First, they may have side effects by modifying user-provided hyperparameters. For example, Zeus adjusts the batch size of a training job which can potentially compromise statistical efficiency, even with optimally-tuned learning rates [24]. This issue becomes particularly challenging in non-recurring DNN training jobs where finding the batch size and learning rate pairs that maintain statistical efficiency is difficult. Second, it is difficult to determine how much performance degradation is acceptable at the cost of saving energy. Typically, the completion time of a training job varies and is unpredictable. Therefore, ML practitioners may not know how much delay they can accept. Furthermore, in a long-running training task, even a small performance degradation implies a significant delay. For instance, 10% degradation of a month-running task translates to three days. Third, these approaches primarily focus on individual GPU training jobs and do not adequately address the energy consumption associated with large-model training. Large model training utilizes multiple GPUs across multiple nodes with various parallelism techniques.

This work proposes ENVPIPE[1], a new energy-saving DNN training framework. ENVPIPE focuses on large model training using multiple GPUs with pipeline parallelism. ENVPIPE addresses the limitation of previous approaches with the design goals: *No accuracy and performance degradation*. With the goals, users can run any DNN training jobs as if they run them without ENVPIPE while saving energy under the hood. To preserve the original accuracy, ENVPIPE does not modify any user-provided hyperparameters such as batch size and does not change data dependency while executing pipeline units. ENVPIPE leverages the side-effect-free control knob only, SM frequency, to save energy. To avoid performance degradation, ENVPIPE utilizes pipeline bubbles inevitably occurring when training large models with pipeline parallelism. ENVPIPE selectively lowers SM frequency to reduce the energy consumption of pipeline units. This control stretches the execution time of pipeline units, but ENVPIPE confines the degree of each stretch up to the available slack time of the bubbles, avoiding end-to-end performance degradation.

[1] **Env**elope + **Pipe**line Parallelism.

This design idea is generally applicable to any DNN training job where layer-wise partitioning is feasible, enabling training with pipeline parallelism across a large number of GPUs. However, realizing the design idea is challenging due to the following problems. **i)** To decide the value of SM frequency, EnvPipe needs to know the trend of clock speed and training performance. The trend varies according to GPU hardware, batch size, and the number of layers in a pipeline stage. Offline profiling to obtain this information is impractical. Instead, EnvPipe performs online profiling by sweeping SM frequencies to obtain the energy-saving curve under the given hyperparameters and GPU hardware. **ii)** How to schedule pipeline units decides the amount of bubbles that can be exploited. It is essential to ensure that a bubble exists after a pipeline unit of which execution time is stretched while ensuring that the next unit, which has data dependency, is sufficiently distant to avoid overall performance degradation.

EnvPipe is implemented as a library using the existing ML framework. The EnvPipe policy and mechanism are clearly separated, so developers can easily add required APIs to support a new ML framework. The current prototype of EnvPipe is implemented on the DeepSpeed [25] library and uses the existing GPU device driver to control SM frequency. We evaluate EnvPipe in real-world workloads: BERT, GPT, Megatron, and ResNet, and demonstrate the performance and energy saving on two GPU hardware: V100 and RTX3090. We perform evaluations of the workload in a single node (4 GPUs) and in multiple nodes (16 GPUs) and show that EnvPipe saves energy up to 25.2% and 28.4% in single-node and multi-node setups respectively while keeping performance degradation to less than 1%.

This paper makes the following contributions:

- We present the design of EnvPipe, a performance-preserving energy-saving DNN training framework that supports distributed training across multiple GPUs.
- EnvPipe preserves the original statistical efficiency by not modifying any user-provided hyperparameters and controls only the side-effect-free control knob.
- We demonstrate EnvPipe saves up to 25.2% and 28.4% energy saving in single- and multi-node GPU servers with less than 1% performance degradation.

The source code of EnvPipe is available on https://github.com/casys-kaist/EnvPipe.

2 Background

2.1 Large Model Training with Parallelism

Recent advancements in language models have focused on increasing the number of parameters, achieving impressive results on various challenging tasks such as language understanding, generation, and reasoning. Google's Pathways Language Model (PaLM), a 540 billion parameter model stacked up with numerous transformer decoder layers, has shown breakthrough results outperforming finetuned state-of-the-art models on various natural language tasks [8]. However, scaling up the model size comes with a cost of increased memory footprint, making it challenging to fit on a single GPU memory, even with the latest GPU like the NVIDIA H100 with 80GB. To efficiently train extremely large models, there have been several efforts to combine various parallelism techniques such as *data*, *tensor*, and *pipeline* parallelism [1, 11, 15, 16, 18–20, 30]

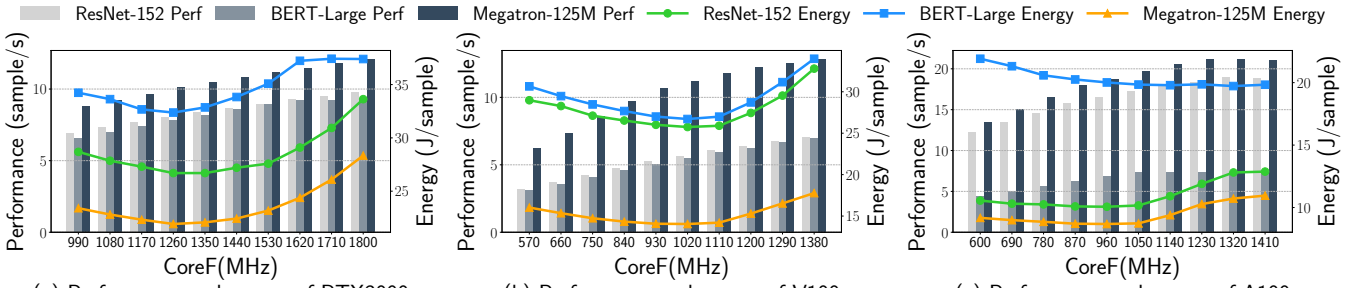
In this study, we focus on the pipeline parallelism [11, 15, 16, 18–20] which is a commonly used technique in training large DNNs whose models cannot fit on a single GPU. With pipeline parallelism, a model is vertically partitioned as evenly as possible to each worker (e.g., GPU) as pipeline stages. For transformer-based models such as GPT, each pipeline stage can have the same number of transformer decoder layers, balancing the execution time across the pipeline stages. To increase pipeline efficiency, the input batch is partitioned into multiple microbatches, and each worker handles the microbatches in a pipelined manner. There are two different approaches to synchronizing model parameters: synchronous and asynchronous. Synchronous pipeline parallelism (S-PP) ensures strict weight update semantics by periodic pipeline flushes. S-PP does not compromise the model's convergence but inevitably incurs pipeline bubbles which lower the training throughput [2]. Asynchronous pipeline parallelism (A-PP) relaxes weight update semantics and fully utilizes the pipeline throughput in a steady state by continuously pipelining microbatches without any pipeline flushes. A-PP hurts the statistical efficiency of the model and can fail to converge to the target accuracy [5].

In this work, we target S-PP which preserves the original statistical efficiency with strict weight update semantics. Due to pipeline flushes after every training iteration, pipeline bubbles are inevitable which lowers the pipeline throughput. Previous studies [15, 16, 20] focused on reducing the bubbles in S-PP. Rather than perceiving bubbles as an obstacle that slows down training, we consider pipeline bubbles as an opportunity to save energy in large model training.

2.2 Energy Scaling Valley Trend in GPUs

As it is convenient for ML practitioners to make use of GPUs rather than NPU-like accelerators for DNN workloads, modern cloud and data centers are operating a huge number of GPUs. However, when training DNN workloads, GPUs incur a significant fraction (e.g. about 70% according to [10]) of total power consumption in the whole system including other components such as CPU and DRAM. This high power consumption of GPUs during DNN training underscores the importance of optimizing their energy efficiency to reduce the overall energy consumption of cloud and data centers. To save energy in GPUs, previous studies [3, 7, 9, 12, 17, 27–29] have utilized *Dynamic Voltage and Frequency Scaling*

[2] GPU remains idle in pipeline bubbles



(a) Performance and energy of RTX3090 (b) Performance and energy of V100 (c) Performance and energy of A100

Figure 1: Energy scaling valley trend in modern GPUs

(DVFS) and focused on balancing performance and energy efficiency. DVFS is a widely studied technique in traditional CPUs to balance performance and energy consumption by scaling voltage/frequency in CPU cores. Generally, scaling down voltage/frequency saves energy but inevitably degrades performance, and GPUs tend to show a more complex energy scaling trend. Tang *et al.* [27] studied the energy scaling trend of various DNN training on modern datacenter GPUs and showed that energy saving is maximized on the middle-level core frequency and the energy consumption curve shows a valley trend when scaling core frequency.

We measure the training throughput and energy consumption by scaling the SM frequency in three different GPUs with various up-to-date DNN models in a single GPU training scenario. We use NVML [21], a library provided by NVIDIA, to adjust the SM frequency of GPUs. NVML can set the maximum limit of SM frequency and monitor the current energy consumption. Figure 1 exhibits that energy consumption decreases when lowering the frequency, but from the middle (e.g., 1350MHz and 1020MHz in RTX3090 and V100), this trend changes oppositely. This is because the training time is prolonged with lower SM frequency. Since energy consumption is related to both current power usage and overall execution time, if the end-to-end execution time increases at a faster rate than the rate of decrease in current power usage, the overall energy consumption increases. Thus, it is crucial to find the optimal point of SM frequency to achieve energy saving since the frequency cannot be lowered below the optimal point.

3 Energy-efficient DNN Training

This section describes the energy-saving problem of DNN training and discusses key insights that motivate the design of our system.

3.1 Objective and Constraints

DNN training is a complex and time-consuming process that places a significant emphasis on achieving statistical efficiency. Consequently, developing energy-efficient strategies for DNN training is a challenging task that can potentially lead to unintended side effects. In this section, we highlight several constraints that are crucial for ensuring the robustness of an energy-saving approach and mitigating any undesirable side effects.

No accuracy degradation. Given that training jobs are often already hyperparameter-searched, we do not modify any user-provided hyperparameters to ensure that the final accuracy after finishing the training is not compromised. Therefore, the way to achieve energy saving in this work is in sharp contrast to prior work that reduces energy consumption by changing hyperparameters, which can affect the final converged accuracy. For example, Zeus [29] studies how different batch sizes affect energy consumption when combined with a wide variety of GPU power scaling levels. The optimal combination chosen in Zeus thus alters depending on hardware and energy efficiency, which is the immediate consequence of the batch size in use. Optimizing energy consumption in this way is advantageous when users issue recurring DNN training jobs or can provide a set of batch sizes and corresponding hyperparameters that promise model convergence regardless of choice. Our target scenarios do not have that expectation from users. So, we decide to use control knobs that preserve the training's original statistical efficiency, such as controlling GPU SM frequency and dependency-aware pipeline scheduling.

No performance degradation. Given that curbing SM frequency to save energy affects DNN training speed, e.g., time taken to execute a single pipeline unit, we do not want to slow down the end-to-end training performance in exchange for energy savings for several reasons. First, from an ML practitioner's standpoint, it is difficult to determine how much performance degradation is acceptable across a wide range of training jobs. The completion time of a training job typically varies and remains unpredictable until the training is completed. Additionally, even a minor performance degradation in a long-running training task can result in a significant delay. For instance, a 10% slowdown in training time might seem small, but it can translate to a three-day increase in the duration of a month-long training job. Second, from a system administrator's standpoint, it is difficult to estimate an abrupt increase in GPU requests caused by prolonged DNN training jobs, which are already computationally expensive. Because GPU resources are highly contended and shared, prolonged DNN training jobs that occupy GPUs for extended periods contribute to increased GPU contention, necessitating the allocation of additional GPUs to alleviate resource bottlenecks.

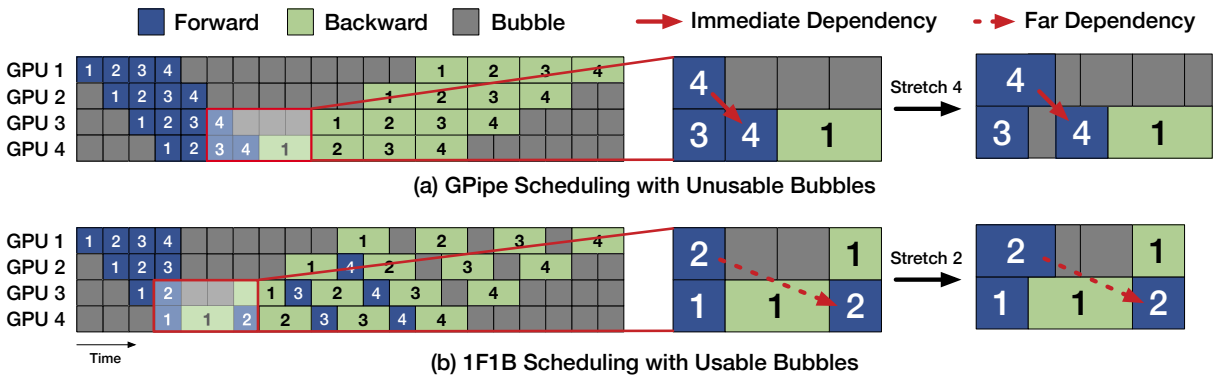


Figure 2: Comparison of two representative pipeline scheduling

Prior systems have centered on navigating the energy-performance tradeoff while keeping energy and time minimum, i.e., Pareto optimality. Unlike these approaches, our work primarily targets saving energy without sacrificing training time when training large models across multiple GPUs with pipeline parallelism, one of the most commonly used techniques in training large DNN models.

3.2 Insights

Our baseline model for distributed training, synchronous pipeline parallelism (S-PP), inevitably incurs bubbles, as shown in Figure 2. We selectively leverage these bubbles only when the pipeline units preceding them do not have immediate data dependencies. By reducing the SM frequency in these units, we can extend their execution time without necessarily delaying the start-up time of the subsequent units that rely on their outputs. This approach enables us to achieve energy savings while preserving performance. However, it is important to constrain the extent of stretching in each unit to a certain limit. This constraint ensures that no adverse performance delays occur as a result of elongated pipeline units.

Usable and unusable bubbles. Several S-PP designs have been proposed to schedule pipeline units during forward-backward computations of a single training iteration. Figure 2 shows the execution details of two representative S-PP designs, `GPIPE` and `1F1B`. The examples take four microbatches on four GPUs. In both cases, each microbatch execution goes through GPUs in order (GPU1 $\rightarrow$ GPU4) during the forward pass (FP) and then in reverse order (GPU4 $\rightarrow$ GPU1) during the backward pass (BP).

We observe that the performance-preserving energy-saving opportunity differs significantly in these two S-PP examples. To better understand this, we classify bubbles into two types: `Unusable` and `Usable`. A bubble is considered unusable when a stretched pipeline unit delays the overall training time. In Figure 2(a), stretching the forward pipeline unit of microbatch 4 (denoted as FP4) in GPU3 delays the start-up time of FP4 in GPU4 because of the `immediate dependency` caused by sending activation. This control delays the overall execution of the total pipeline. Even though plenty of bubbles are available after FP4 in GPU3, these bubbles are considered unusable, and exploiting unusable bubbles to save energy slows down training throughput, violating the constraints defined in § 3.1.

On the contrary, a bubble is considered usable when a stretched pipeline unit does not postpone the overall training time. For example, in Figure 2(b), stretching the forward pipeline unit of FP2 in GPU3 does not affect the execution of the backward pipeline unit of microbatch 1 (denoted as BP1) in GPU4 as these two units do not exhibit data dependency. FP2 in GPU4 exhibits the data dependency for activation communication, but it begins execution much later. We refer to this type of dependency as `far dependency`. Consequently, when utilizing the bubbles after FP2 in GPU3 for energy saving, none of the pipeline units in GPU4 gets penalized.

Based on this observation, we seek to exploit as many usable bubbles as possible for maximizing energy saving without performance degradation.

4 Design

4.1 Design Overview

This section presents the overview of our proposed system called EnvPipe. For a given DNN model and its hyperparameters, EnvPipe automatically tunes the order of pipeline units and generates an energy-saving plan controlling the SM frequency without any manual efforts from users. First, EnvPipe profiles the energy consumption of the DNN training job for each pipeline stage to understand performance and energy tradeoffs. Second, to increase the energy-saving opportunities, EnvPipe reschedules the pipeline units elaborately, increasing the amount of usable bubbles without breaking any data dependencies between the pipeline units. Last, EnvPipe finds an optimal SM frequency for pipeline units on the non-critical path to maximize energy savings without sacrificing training throughput.

Figure 3 depicts the overview of EnvPipe. EnvPipe consists of `online profiler` (§ 4.2), `scheduler` (§ 4.3), `frequency planner` (§ 4.4), and `execution engine`. At its

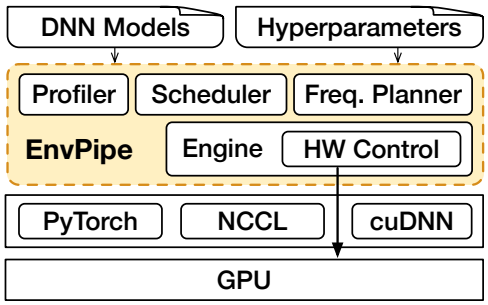


Figure 3: EnvPipe Overview

core, EnvPipe design clearly separates policy from mechanism. EnvPipe generates a performance-preserving energy-saving plan by executing in the workflow of online profiler → scheduler → frequency planner (policy). The execution engine runs the energy-saving plan (mechanism).

Policy: Building the energy-saving plan. At first, EnvPipe runs the `online profiler` to construct the energy valley curves for each pipeline stage when training the given DNN model. Based on the online profiling result, EnvPipe `scheduler` decides the best schedule of forward and backward execution units, which creates plenty of usable bubbles. Using the scheduling decision, EnvPipe runs the `frequency planner`. It lowers the SM frequency of all pipeline units inside the outermost path of the total pipeline (we call it *envelope*) to the optimal value identified from the energy valley curve. At this point, it is likely to degrade the training performance because execution units inside the envelope are stretched. To avoid performance degradation, EnvPipe identifies the performance-critical path and reconfigures the SM frequency (i.e., undo lowering SM frequency) of all units in the performance-critical path to avoid performance slowdown.

After these steps are completed, EnvPipe obtains the energy-saving plan that specifies I) a schedule (placement) of forward and backward pipeline units, and II) SM frequency value of each pipeline unit, which achieves energy saving without degrading performance.

Mechanism: Executing the energy-saving plan. The execution engine provides APIs for the `online profiler` and the `frequency planner`. Internal APIs used in the EnvPipe's execution engine are translated to ML platform-specific APIs (e.g., PyTorch API calls). The engine includes HW control APIs communicating GPU device driver to control SM frequency.

EnvPipe is implemented as a user-level library to invoke APIs of underlying ML platforms, providing an easy-to-use, platform-independent way to control multi-GPU pipeline scheduling and energy consumption. In addition, due to this clean separation of policy and mechanism, EnvPipe can be applicable to any ML platform by implementing required APIs in the execution engine to support the ML platform [3].

[3]The current implementation supports PyTorch only

4.2 Fine-grained Online Profiling

As shown in Figure 1, energy consumption shows the valley trend according to SM frequency. The form of valley curves depends on GPU hardware, batch size, model architecture, and the method of splitting the model for pipelining. Therefore, EnvPipe runs the *online* profiler to obtain the energy valley curve from given DNN models, GPU hardware, and hyperparameters. For each pipeline stage, the online profiler sweeps available ranges of SM frequency and measures energy consumption to find the optimal SM frequency that maximizes energy saving. The profiling steps are seamlessly integrated into the training procedure, allowing EnvPipe to continue training with the weight version obtained from the profiling steps. The energy valley curves for each pipeline stage are generated within 100 steps and just 5 steps per frequency are enough to detect the optimal point where the energy-saving trend changes oppositely. Since training a model usually requires thousands to millions of steps, the overhead of the online profiler can be considered negligible. Note that the SM frequency of a pipeline unit cannot be lowered below the optimal energy-saving point which stretches the execution time of the pipeline unit to about 20 - 25% in our GPU settings. In addition, the online profiler measures the execution time at maximum and optimal SM frequency's forward and backward pass, and available GPU memory, which is used in the scheduling phase.

4.3 Scheduler: Utilizing Bubble

Design problems. As discussed in § 3.2, the scheduling of forward and backward pipeline units determines the amount of usable bubbles. When making scheduling decisions, it is important to consider two key questions: 1) how to effectively identify usable and unusable bubbles, and 2) how to optimize the utilization of usable bubbles by scheduling pipeline units.

Identifying usable bubbles. To answer the first question, we first need to identify pipeline units in the performance-critical and non-critical paths. Figure 4(a) shows bubbles and the performance-critical path (red boxes). It is important to note that usable bubbles are placed after pipeline units of the non-performance-critical paths. On the contrary, bubbles after the pipeline units of performance-critical paths are unusable. For example, if we stretch BP8 in GPU3 (★) which is on the performance-critical path to use the following bubbles, it will delay the start of BP8 in GPU2 (♠) causing performance degradation in the overall pipeline execution.

Optimizing utilization of usable bubbles. To answer the second question, we should consider the stretch limit of the pipeline units. Recall that there is a limit for the pipeline unit to get stretched because EnvPipe does not set SM frequency below the optimal point, which is usually about 20 – 25% (§ 4.2). To optimize the utilization of usable bubbles, EnvPipe should distribute the usable bubbles since a certain group of bubbles at the front of the pipeline with a long idle

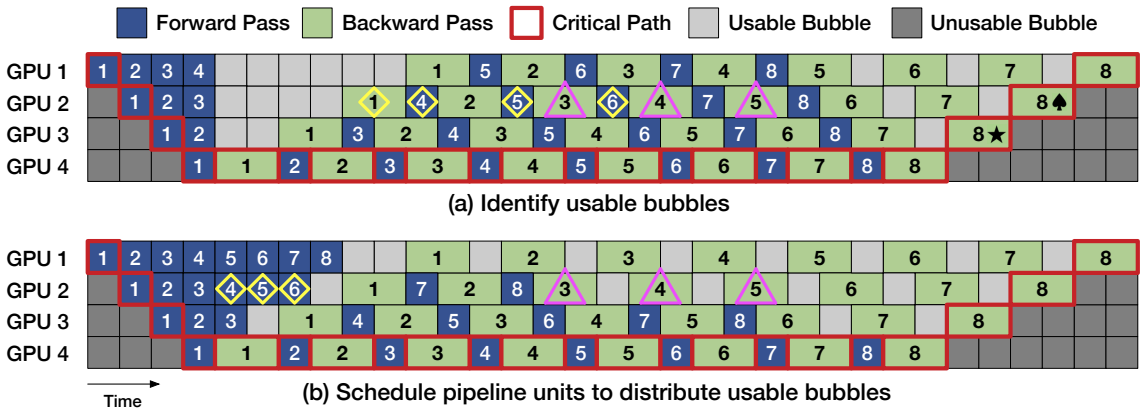


Figure 4: EnvPipe scheduling

time can be underutilized. For pipeline units to utilize the usable bubbles at their best, the usable bubble must be evenly distributed throughout the pipeline execution.

The EnvPipe scheduler operates in two phases: initialization and rescheduling. During initialization, EnvPipe employs a proven method for scheduling pipeline execution, which has been successfully deployed in a wide range of environments. Next, in order to enhance the utilization of usable bubbles, EnvPipe scheduler reschedules pipeline units in a manner that evenly distributes these bubbles while preserving the original data dependency between pipeline units.

Initialization. Among various existing approaches, we select one of the known methods that have the fewest units on the performance-critical path because it contains more usable bubbles. As observed in § 3.2, for the 1F1B schedule, pipeline units only along the outermost path (denoted as *envelope*) are on the performance-critical path. Therefore, we select the 1F1B schedule, which has the minimum number of pipeline units on the performance-critical path, as a starting point and further reschedules the pipeline units based on this initialization.

Rescheduling pipeline units. After initialization, EnvPipe reschedules pipeline units in order to reserve usable bubbles right behind pipeline units and to distribute usable bubbles among pipeline units. To save energy consumption, EnvPipe can stretch pipeline units up to the slack time made by the following bubble. Figure 4(b) shows the result of scheduling Figure 4(a). EnvPipe moves FP units to upfront usable bubbles (e.g., FP4, FP5, and FP6 in GPU2 ◇), generating usable bubbles after backward units (e.g., BP3, BP4, and BP5 in GPU2 △).

EnvPipe can compute how many FPs can be rescheduled and stretched by computing the available slack time of bubbles. When rescheduling the FP units, EnvPipe considers the following conditions. I) EnvPipe never breaks the data dependency for sending and receiving activations and gradients. For instance, FP3 in GPU3 starts only after the activation is sent from FP3 in GPU2. This is essential for preserving the original data dependency of the pipeline execution. II) Because EnvPipe moves forward units upfront, it needs to hold additional activation generated by each forward unit, which uses extra GPU memory. The memory used by activation is freed after the corresponding backward unit consumes it. The size of memory used by each activation is obtained by the online profiler. Therefore, EnvPipe computes available memory to hold the activations and only reschedules a certain number of forward units that can fit in the available memory to avoid an out-of-memory error.

Algorithm 1 Frequency Planner

```
while True do
    ExecutePipelineStep()
    criticalPath ← FindCriticalPath()
    if criticalPath ≠ outer envelope of total pipeline then
        ReconfigureCriticalPath(criticalPath)
    else
        break
    end if
end while
```

Figure 5: Frequency Planner Algorithm

4.4 Frequency Planner: Minimizing Performance Impact

Design problems. According to the scheduling decision, the goal of the frequency planner is maximizing energy saving while minimizing performance impact (less than 1%) by controlling SM frequency. To achieve this, the system leaves the SM frequency at its maximum for units on the performance-critical path and lowers the frequency to its energy-optimal value (obtained from the online profiler) for units not on this path using the available slack time of usable bubbles. However, this does not guarantee minimal performance degradation, since not all bubbles can accommodate the stretched pipeline units inside the envelope. So the question is how to selectively reconfigure the SM frequency to minimize performance impact. In addition, when reconfiguring the frequency of units on the performance-critical path, it is possible that the path may change. To prevent performance slowdowns, the system must be able to efficiently identify the new performance-

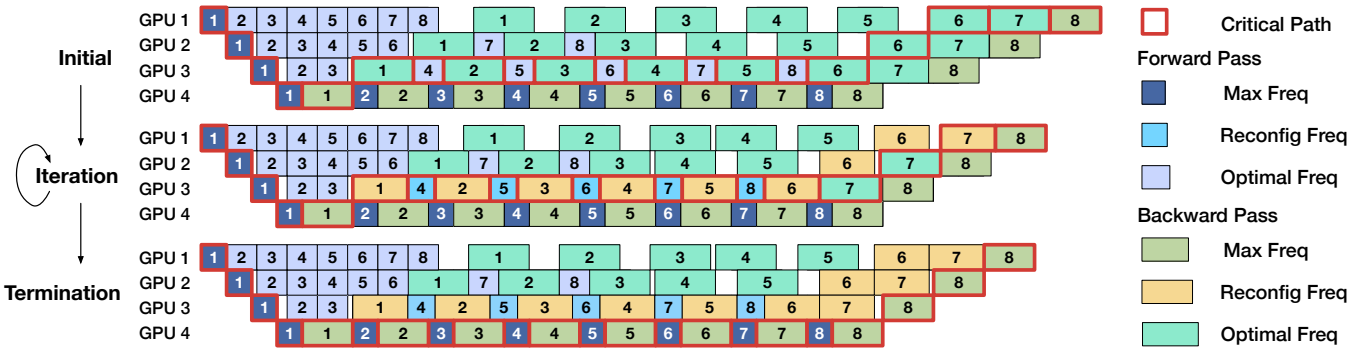
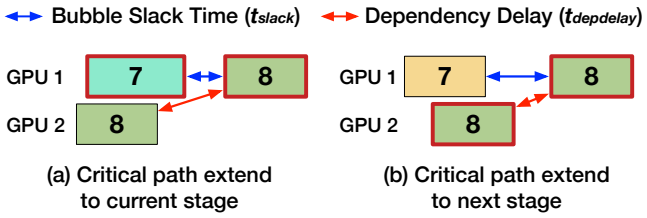


Figure 6: ENVPIPE frequency planning

critical path. This leads to the second design question: how can the performance-critical path be identified quickly and effectively?

Our strategy. Initially, ENVPIPE sets the energy-optimal SM frequency to pipeline units on non-critical paths which is all pipeline units inside the outer envelope of the total pipeline. To achieve the design goal, ENVPIPE runs an iterative algorithm as shown in Figure 5. Figure 6 illustrate the steps. After executing a single pipeline step, at Line 3, the algorithm finds the critical path of the executed pipeline (**Initial** of Figure 6). Then, at Line 5, it reconfigures SM frequency of pipeline units on the detected performance-critical path to avoid performance slowdown (**Iteration** of Figure 6). It repeats the *find and reconfigure* steps until the performance-critical path becomes the outer envelope of the pipeline. The algorithm stops when the critical path is identical to the outer envelope (**Termination** of Figure 6). The termination condition of the iterative algorithm ensures that the performance-critical path is the same as that of running pipeline parallelism without using ENVPIPE while our frequency planner stretched as many pipeline units as possible on the non-critical path.

Algorithm for finding the critical path. Figure 7 shows the algorithm to identify the performance-critical path. The algorithm incrementally builds the critical path backward. *current* in Figure 7 is a cursor, pointing to pipeline units, and moves backward. Initially, the cursor starts with the last unit (BP8) in GPU1. The algorithm repeatedly updates the cursor to point to the next pipeline unit to insert into the critical path until the cursor reaches the starting unit (FP1) in GPU1. At line 9, the algorithm decides the next pipeline unit to extend the critical path backward. Figure 7(a) and (b) illustrates the idea. Recall that the critical path consists of pipeline units that delay the overall execution time when stretched. Thus, we should find the pipeline unit that is affecting the start time of the pipeline unit that the cursor is currently pointing at. Let's assume the current cursor points to BP8 in GPU1. The algorithm finds the pipeline unit that is affecting the start time of BP8 in GPU1 by comparing the slack time — the bubble between BP7 and BP8 in GPU1 (t_{slack} of *current*) — and the dependency delay — spare time between the end of BP8 in GPU2 and the start of BP8 in GPU1 ($t_{depdelay}$). If t_{slack} of



Algorithm 2 Finding critical path

1: *GPUID*: e.g., GPU 1 - GPU 4
2: t_{slack}: Slack time of the precedent bubble
3: $t_{depdelay}$: Delay between *pipelineUnits* with data dependency
4: *current* ← last backward *pipelineUnit* in GPU 1
5: *criticalPath*.insert(*current*)
6: **while** *current* ≠ first forward *pipelineUnit* in GPU 1 **do**
7: $n \leftarrow$ *GPUID* of *current*
8: $k \leftarrow$ *microbatchID* of *current*
9: **if** t_{slack} of *current* $< t_{depdelay}$ **then**
10: *current* ← the previous *pipelineUnit* in GPU n
11: **else**
12: **if** *current is* forward *pipelineUnit* **then**
13: *current* ← *pipelineUnit* k in GPU $(n-1)$
14: **else**
15: *current* ← *pipelineUnit* k in GPU $(n+1)$
16: **end if**
17: **end if**
18: *criticalPath*.insert(*current*)
19: **end while**

Figure 7: Finding critical path

current $< t_{depdelay}$ (Figure 7(a)), the precedent pipeline unit of the same GPU is affecting the start time of BP8. So the algorithm updates the cursor to BP7 in GPU1 to add to the critical path. Otherwise, the pipeline unit that has the data dependency from the next GPU is affecting the start of BP8. Thus, the cursor is updated to BP8 in GPU2 and is added to the critical path. After that, the algorithm repeats the same step iteratively. The algorithm ends when the cursor reaches the first forward unit in GPU1, which is the first pipeline unit on the critical path that decides the starting time of the overall pipeline execution step.

Algorithm for reconfiguring critical path. EnvPipe reconfigures the pipeline units on the critical path by increasing a small amount of SM frequency. The reconfiguration should be done in small steps in an iterative way for two reasons. First, by increasing the frequency in small increments, the stretched execution time can be gradually shortened, likely finding the point of fully utilizing the slack time of the usable bubbles. Second, due to the complex data dependencies between pipeline units, the critical path may change after reconfiguring pipeline units on the critical path. By reconfiguring in small steps, it will be less likely to unnecessarily increase the frequency of pipeline units on the non-critical path, since the critical path may have changed during the reconfiguration process.

The key question in the algorithm is identifying which pipeline units on the critical path should be reconfigured. To determine this, we use the observation of the trend in the energy-scaling valley curve shown in Figure 1. From the optimal point (minimum energy), as the SM frequency increases, the energy consumption and performance increase at different rates. Therefore, EnvPipe defines the *performance-energy utility* as the ratio of performance increase to energy increase for a frequency increase. In general, the performance-energy utility diminishes as it is further from the optimal point. Thus, to maximize the utility, the system should prioritize reconfiguration of SM frequency close to the optimal point.

Using this observation, EnvPipe takes a `balanced` approach. After finding the critical path, EnvPipe finds the pipeline units with the minimum SM frequency (i.e., closest one to the optimal value) and increases their SM frequency. This allows for the SM frequencies of all pipeline units on the critical path to be balanced as much as possible. For comparison, the system also implemented a simple approach called `greedy`. This approach selects pipeline units backwards from the end of the critical path and reconfigures them until their frequency is the maximum default frequency of a GPU. This approach also achieves the performance goal by increasing the SM frequency of pipeline units on the performance-critical path, but not in a balanced way.

4.5 Discussions

4.5.1 Size of Bubble and Energy Saving

The size of the bubble highly influences the achieved energy saving. The opportunity to leverage pipeline bubbles increases as the size of the bubble increases thus leading to higher energy savings. Since EnvPipe does not change the user-provided hyperparameters, achieved energy saving may differ according to the user-provided input or bubble-reducing methods that were studied in previous S-PP works [15, 16, 20].

Number of microbatches. The size of the bubble gets amortized over the number of micro-steps. Thus, as the number of microbatches increases, the fraction of the pipeline bubble decreases. For the portion of the bubble to be minimized, the number of microbatches should be larger than the number of pipeline stages. However, increasing the number of microbatches indefinitely is not possible because increasing the number of microbatches leads to an increase in global batch size. Even with a carefully tuned learning rate, there is a maximum limit in global batch size to preserve the statistical efficiency [24]. We show as a sensitivity study how the number of microbatches affects energy saving.

Partition method of pipeline stages. The partition method to split pipeline stages affects the size of pipeline bubbles. Pipeline bubbles are minimized when the execution time among pipeline stages is well-balanced, but it is not straightforward. Several partition methods are possible, but each of them has its own pros and cons. First, the model can be partitioned by balancing the execution time of layers per stage. By balancing the execution time of layers per stage, the size of the bubble is minimized. However, balancing the execution time of layers may lead to memory imbalance among pipeline stages. Second, stages can be partitioned by balancing the memory consumption of GPUs. Balancing the memory footprint across pipeline stages can provide advantages in memory-constrained environments. However, because of the imbalance of execution time among stages, the size of the bubble increases. We evaluate how the stage partition methods affect the energy saving in § 6.2.3 (Table 2).

Bubble-reducing methods. Our work stands apart from previous studies that aim to reduce bubbles in S-PP [15, 16, 20]. While these studies may reduce pipeline bubbles, the bubbles cannot be completely eliminated, so EnvPipe can still leverage the bubbles to save energy. Moreover, these bubble-reducing methods have inherent drawbacks. For instance, Merak [15] necessitates activation recomputation, leading to additional performance overhead by repeated computations. Chimera [16] proposes a bidirectional pipeline but with additional memory consumption from weight parameters and activations since a single stage needs to maintain weights and activations that were originally maintained by two stages. PTD-T [20] introduces an interleaved 1F1B pipeline schedule, which reduces bubbles but adds communication overhead to the scheduling process. In contrast, our approach in EnvPipe effectively utilizes bubbles without introducing these drawbacks, offering a clear advantage in saving energy in pipeline parallelism.

4.5.2 Energy Consumption of Bubbles

Readers may raise the following question, "Would a naïve approach that just reduces the power consumption of bubbles save more energy than EnvPipe?" Even if one hypothetically assumes that the power usage of bubbles is reduced to 0W, EnvPipe's approach still demonstrates superior energy savings. Figure 8 illustrates a simplified example comparing the naïve approach and EnvPipe. In the naïve approach, the total energy consumption is 8.75J, where the forward unit consumes 8.75J of energy while the bubble consumes 0J. On the

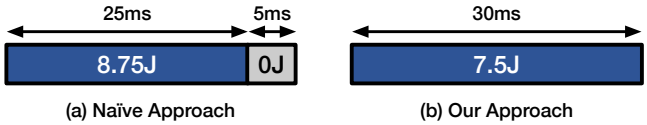


Figure 8: Energy consumption of bubbles

other hand, in EnvPipe's approach where the forward unit is stretched to exploit the bubbles, the total energy consumption becomes 7.5J. The energy consumption of the stretched forward unit is already lower than the naïve approach's forward unit since decreasing the SM frequency reduces the end-to-end energy consumption of the forward unit. As demonstrated in this example, even when the power usage of bubbles is reduced to 0W, EnvPipe's approach still has lower end-to-end energy consumption. Furthermore, in RTX3090, we measure that the power usage of bubbles is approximately 100W out of 350W at the lowest SM frequency with P2 P-State.

4.5.3 Scaling with Data Parallelism

The common practice for training large models that cannot fit in a single GPU is to combine pipeline parallelism with data parallelism or tensor parallelism, allowing for sharded models to fit within the constraints of a single GPU. We show in the evaluation section that our performance-preserving energy-saving approach with pipeline parallelism can easily scale with data parallelism in multi-node training.

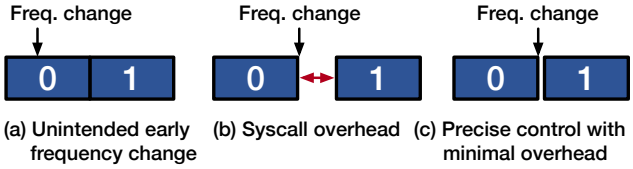
5 Implementation

We implement our prototype, EnvPipe, on top of DeepSpeed [25]. EnvPipe's HW controller to lock SM frequency and DeepSpeed's basic mechanism such as executing forward/backward pass, send/receiving activations and gradients, and reducing computed gradients are used for executing EnvPipe's performance-preserving energy-saving plan. In this section, we introduce some important considerations when utilizing the underlying mechanisms of ML platforms since naïve usage of those mechanisms can lead to ineffective usage of bubbles or performance degradation.

5.1 Asynchronous Communication

Activations and gradients must be transferred between GPUs for the pipeline to be executed. When sending activations or gradients, communication needs to be asynchronous (i.e. non-blocking) for pipeline units to effectively use bubbles. If GPU communication is synchronous, idle time of bubbles is wasted since blocking communication calls prevents the next scheduled pipeline units to execute. When the source GPU sends data to the target GPU, the communication gets blocked until the target GPU receives the data and the next scheduled pipeline units cannot be executed, thereby wasting the opportunity to utilize pipeline bubbles. On the other hand, for non-blocking asynchronous communication, when the source GPU sends data to the target GPU, the source GPU does not have to wait until the target GPU receives the data and can execute the next pipeline unit, effectively utilizing pipeline bubbles.

NCCL's default blocking p2p communication can be easily changed to non-blocking communication by increasing the NCCL buffer size with NCCL_BUFFSIZE environment variable. NCCL buffer is used when communicating data between pairs of GPUs. P2p send operation fills up the target GPU's buffer and the target GPU fetches data from the buffer in FIFO for another send operation to fill the buffer. If the NCCL buffer is full, send operation should wait until the buffer of target GPU has free space. If the NCCL buffer has enough free space, p2p send operation can complete without waiting for p2p recv operation to be called from target GPU. EnvPipe makes sure that NCCL buffer size is enough to handle all activations and gradients to be communicated in a non-blocking way to effectively use bubbles.




```
# (a) Unintended early frequency change
exec_forward_pass(0)
lock_gpu_clock(1300)
exec_forward_pass(1)
```

```
# (b) System call overhead
exec_forward_pass(0)
torch.cuda.synchronize()
lock_gpu_clock(1300)
exec_forward_pass(1)
```

```
# (c) Precise control with minimal overhead
exec_forward_pass(0)
torch.cuda.synchronize()
threading.Thread(lock_gpu_clock, args=(1300))
exec_forward_pass(1)
```

Figure 9: Precise SM frequency control

5.2 Precise SM Frequency Control

Controlling SM frequency from user space needs an `ioctl` system call to the device driver which can induce overhead lowering the training throughput. Also because of the asynchronous CUDA programming model, `ioctl` calls on the CPU side should be executed with precise synchronization barriers between pipeline units for SM frequency to be controlled at exact timing.

Figure 9 shows how precise SM control with minimal overhead can be performed. In Figure 9(a), `lock_gpu_clock()` is called between two forward executions without any synchronization barrier. Because of the asynchronous CUDA programming model, `ioctl` call gets executed on the CPU side before the first forward execution completes, resulting in an unintended early frequency change. In Figure 9(b), the synchronization barrier is placed before `lock_gpu_clock()`,

	Model	Microbatch	Minibatch
Single-V100	BERT-336M	4	64
	GPT-125M	2	32
	Megatron-125M	4	64
	ResNet-152	2	32
Single-3090	BERT-1.3B	4	64
	BERT-3.9B	2	32
	GPT-350M	4	64
	Megatron-350M	4	64
	Megatron-760M	4	64

Table 1: Model configuration for single-node training.

preventing unintended early frequency change. However, the `ioctl` call on the CPU side with the default synchronous programming model delays subsequent forward execution calls and GPU remains idle until the `ioctl` call returns. In Figure 9(c), by executing `lock_gpu_clock()` in another thread, subsequent forward execution on the GPU side can be concurrently executed with the `ioctl` call on the CPU side, performing precise frequency control with a minimal performance impact.

6 Evaluation

Our evaluations focus on confirming our energy-saving design choices that preserve DNN training performance. Specifically, we compare performance and energy consumption for various scheduling strategies when training different, up-to-date DNN models, including transformer-based language models and CNN models, on both single-node and multi-node training setups. To faithfully demonstrate the benefits of our approach in challenging scenarios, we have all models partitioned to balance execution time among pipeline stages, e.g., balancing the number of layers for transformer-based models. This pipeline-balanced parallel execution provides high training performance, making the energy savings on top of it more meaningful.

6.1 Methodology

Testbed setup. All experiments are performed on PyTorch 1.13. For single-node experiments, we use two different server setups: Single-V100 and Single-3090. **Single-V100** uses AWS `P3.8xLarge` instance which has four NVIDIA Tesla V100 GPUs with 16GB of memory each, Intel Xeon E5-2686 v4 CPU (32 vCPUs), and 244GB of main memory, while **Single-3090** has four NVIDIA Ampere RTX3090 GPUs with 24GB of memory each, Intel(R) Xeon(R) Gold 6326 CPU (64 vCPUs), and 256GB of main memory. For multi-node experiments, our setup **Multi-V100** uses two AWS `P3.16xLarge` instances each with 8 NVIDIA Tesla V100 GPUs, Intel Xeon E5-2686 v4 CPU (64 vCPUs), and 488GB of main memory. So, Multi-V100 is equipped with a total of 16 V100 GPUs. These cloud instances are connected to a 25Gbps Ethernet network.

Benchmarks. We compare EnvPipe with the following energy-saving methods:

- Baseline: run all GPUs with maximum SM frequency.
- Uniform: run all GPUs with optimal SM frequency that represents the minimum point in the energy valley curve.
- NoRecfg: EnvPipe without reconfiguring critical path to minimize performance impact.

Our single-node experiments are based on nine different models, as shown in Table 1. All models run with 16 microbatches, while for Megatron-125M, ResNet-152, and BERT-3.9B, the minibatch and microbatch size are reduced to fit in GPU memory. For multi-node experiments, we combine data parallelism (DP) and pipeline parallelism (PP) in several different ways across 16 GPUs. The number of microbatches for pipelining changes according to the DP and PP dimensions while the size of minibatch remains constant.

Metrics. Performance for all experiments is measured by averaging the throughput of 30 training iterations after warm-up and energy consumption is measured using the NVML library [4] during the 30 training iterations.

6.2 Experimental Results

6.2.1 Single-node Energy Saving

Main results. We first compare EnvPipe to Baseline, which consumes the most energy among competing methods, to highlight our ability to preserve performance. We measure the throughput and energy consumption for the benchmarks in Table 1 and present the results in Figure 10. The results show that EnvPipe consistently uses less energy than Baseline while retaining the original training performance for all models, confirming its effectiveness. Specifically, for Single-V100, EnvPipe saves energy consumption by an average of 18.6%, ranging from 12.1% to 25.2%. For Single-3090, EnvPipe saves energy consumption by an average of 12.8%, ranging from 8.1% to 19.4%. Performance is mostly preserved, with throughput only degrading less than 1% in all cases.

Analysis of energy saving. We next compare EnvPipe to other baselines by focusing on two models, BERT-1.3B and GPT-350M, trained on Single-3090. The results from Figure 11 show that other naïvely designed strategies, such as Uniform or NoRecfg, fail to meet our energy savings goal without sacrificing performance. Specifically, in Uniform, as all GPUs blindly scale SM frequency without considering performance effects, both throughput and energy consumption inevitably become the lowest. In NoRecfg, which does not reconfigure the SM frequency, all pipeline units inside the outer envelope are stretched to the largest extent. This strategy also inevitably degrades performance since not all bubbles can accommodate those stretched pipeline units inside the envelope. On the other hand, EnvPipe allows for more adaptive and effective use of pipeline bubbles, minimizing the impact on performance.

[4] Energy consumption of GPUs (not entire system).

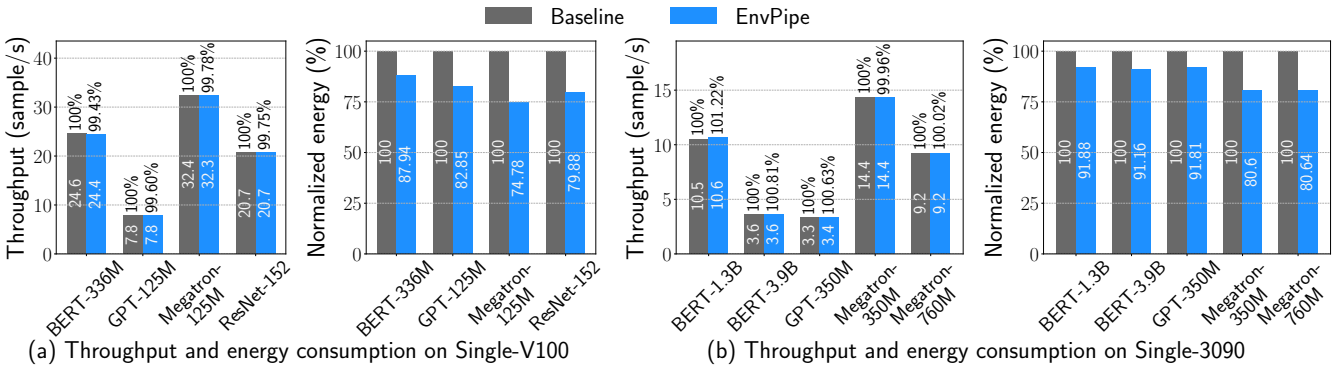


Figure 10: Throughput and energy consumption of various DNN models in single-node training

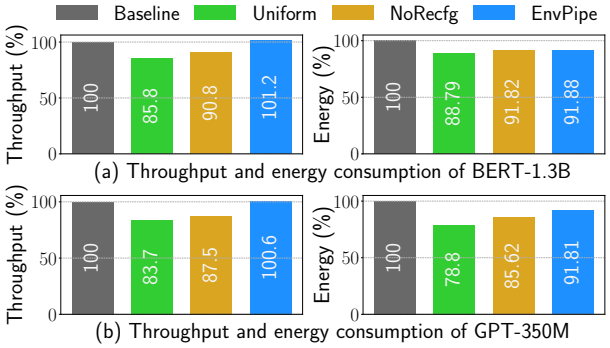


Figure 11: Normalized throughput and energy consumption breakdown

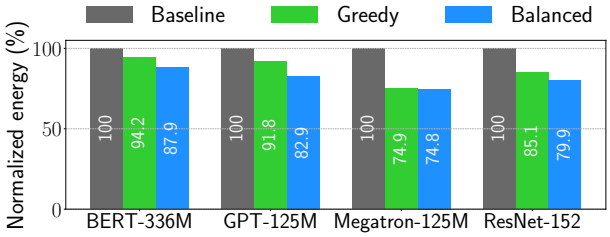
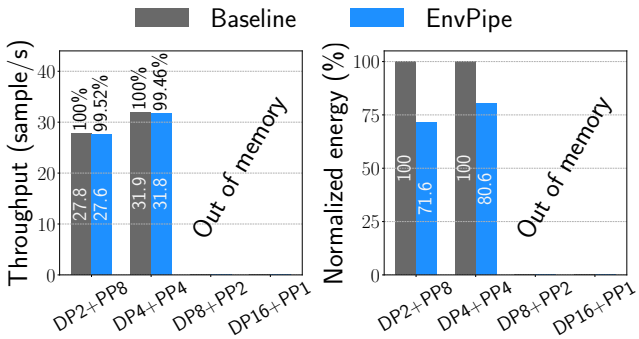


Figure 12: Comparison of reconfiguration policy

Comparison of reconfiguration policies. We now evaluate two different reconfiguration policies for the frequency planner of EnvPipe, balanced (our default) and greedy, using models trained on Single-V100. The results shown in Figure 12 demonstrate that for BERT-336M, GPT-125M, and ResNet-152, the balanced method saves more energy than the greedy method, with a range of 5.2 to 8.9%. However, the difference in energy savings is insignificant for Megatron-125M. This is because the portion of pipeline bubbles (a measure of how much time is spent by bubbles) is larger for Megatron models due to additional computation on the loss computation layer, which is insignificant in other models, leading to longer execution times in the last stage of the pipeline. This portion exists even though the model is evenly partitioned across GPUs w.r.t. the number of transformer-based layers placed on each GPU. In summary, the balanced method delivers more benefits than the greedy method but exhibits different relative effectiveness according to how bubbles are composed.



DP	PP	Microbatch	Minibatch	Num of Microbatch
2	8	2	512	128
4	4	2	512	64
8	2	2	512	32
16	1	2	512	X

Figure 13: Throughput and energy consumption of Megatron-1.3B on Multi-V100

6.2.2 Multi-node Energy Saving

To study the efficacy of EnvPipe under multi-node training, we examine the throughput and energy consumption of Megatron-1.3B trained on Multi-V100. Training sweeps different data parallel (DP) and pipeline parallel (PP) dimensions, as shown in Figure 13. Achieving efficient memory utilization in distributed training can be challenging when relying solely on data parallelism. Due to memory constraints, training could not be completed for DP8+PP2 and DP16+PP1. Also, we omit DP1+PP16, i.e., single-way DP combined with 16-way PP, since it is challenging to evenly split 24 transformer-based layers in Megatron-1.3B over 16 pipeline stages. We thus compare the throughput and energy consumption of Baseline and EnvPipe mainly for DP2+PP8 and DP4+PP4, and show the results in Figure 13. EnvPipe saves energy by 28.4% for DP2+PP8 and 19.4% for DP4+PP4 compared to Baseline. Similar to the single-node experiments, the throughput degradation for both cases is less than 1%, indicating that EnvPipe can maintain its benefits when scaling to two or potentially more GPU nodes.

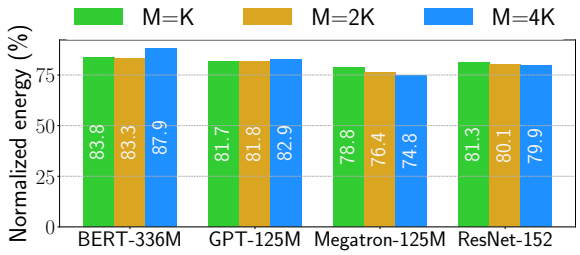


Figure 14: Sensitivity study of number of microbatches (M = number of microbatches, K = number of pipeline stages)

6.2.3 Sensitivity Study

Different number of microbatches. We examine how changing the number of microbatches affects the effectiveness of EnvPipe on Single-V100. The results, shown in Figure 14, reveal that the impact on energy savings differs between Megatron-125M and other models (BERT-336, GPT-125M, and ResNet-152). Megatron-125M model can save more energy when trained on a larger number of microbatches. This is largely due to an imbalance in pipeline stages and a larger portion of pipeline bubbles as compared to other models, which can be effectively harnessed over a long sequence of pipelined executions. However, for other models, increasing the number of microbatches does not result in further energy savings because it quickly leads to amortized bubble sizes.

Stage partition methods. Transformer-based models, which we have used for experiments thus far, are easy to partition in a balanced manner because of their regular structures. However, models like CNNs have non-regular structures, so different partitioning methods with EnvPipe can presumably offer different benefits. To investigate this, we evaluate two different partitioning methods, balancing execution time or memory footprint, using a popular CNN model, ResNet-152 on Single-3090. Both methods are considered beneficial from the perspective of system utilization, as they make training faster or more memory-efficient. The results from Table 2 show that balancing execution time leads to higher throughput, but the energy savings between the two methods are similar. This is because the stages at the front of the pipeline have more activations to store in memory, so shifting computation to the back of the pipeline stage can balance memory usage but cause an imbalance in computation between stages. Consequently, there is not much computation left to stretch in the pipeline stages at the front, limiting energy-saving opportunities.

7 Related Work

Pipeline parallelism. Bubbles in pipeline parallelism have been considered as an obstacle that slows down training throughput and previous studies focused on reducing the bubble with new scheduling methods [15, 16, 20]. On the other hand, EnvPipe considers bubbles as an opportunity and leverages bubbles to save energy.

Data center energy analysis. Recent studies focused on analyzing carbon emission and energy usage to measure the environmental impact when training large models in datacenters [14, 22, 23]. Treehouse [4] aims to reduce the carbon intensity of datacenters from a software perspective by providing suites of resources to application developers to better understand the trade-off between performance and carbon emissions. Strubell *et al.* [26] emphasizes the importance of quantifying the environmental cost of training neural network models for NLP.

Partition Method	GPU Memory (GB)	Perf. (sample/s)	Energy (%)
Execution Time	7.5 / 7.1 / 6.1 / 3.8	20.5	83.1
Memory	6.9 / 5.0 / 6.0 / 5.0	14.2	84.4

Table 2: Comparison of partition method of pipeline stages

Improving energy efficiency with GPU DVFS. Previous approaches to improve energy efficiency in GPUs have utilized GPU DVFS techniques by characterizing the relationship between performance and energy efficiency [6, 9, 13, 28]. Tang *et al.* [27] studied the energy scaling trend of various DNN training jobs on modern datacenter GPUs. Zeus [29] considers batch size as a new control knob for improving energy efficiency on DNN training, navigating the energy-performance tradeoff with Pareto optimality. Similarly, Batchsizer [17] considers batch size on DNN inference. Unlike previous approaches, EnvPipe saves energy by leveraging only the side-effect-free control knob in multi-GPU training.

8 Conclusion

In this study, we propose EnvPipe, a performance-preserving energy-saving DNN training framework. EnvPipe saves energy with no accuracy and performance degradation by leveraging bubbles in pipeline parallelism. EnvPipe profiles the optimal SM frequency of each pipeline stage, schedules pipeline units to make the best out of usable bubbles, and selectively reduces the SM frequency of pipeline units as much as possible with only a certain limit to avoid any adverse performance delay. EnvPipe can save energy up to 25.2% energy in single-node training with 4 GPUs and 28.4% in multi-node training with 16 GPUs with less than 1% of performance degradation.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable comments and feedback. This work was supported by Samsung Electronics, Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-00871, Development of DRAM-Processing-In-Memory Chip for DNN Computing, 50%), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00211901), and Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (23ZS1300).

References

[1] Deepspeed: Extreme-scale model training for everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/. Accessed: 2023-01-12.

[2] Measuring greenhouse gas emissions in data centres: the environmental impact of cloud computing. https://www.climatiq.io/blog/measure-greenhouse-gas-emissions-carbon-data-centres-cloud-computing. Accessed: 2023-01-12.

[3] Yuki Abe, Hiroshi Sasaki, Shinpei Kato, Koji Inoue, Masato Edahiro, and Martin Peres. Power and performance characterization and modeling of gpu-accelerated systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 113–122, 2014.

[4] Thomas E. Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. Treehouse: A case for carbon-aware datacenter software. *CoRR*, abs/2201.02120, 2022.

[5] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, low-cost training of massive deep learning models. In *EuroSys 2022*, April 2022.

[6] Srikant Bharadwaj, Shomit Das, Yasuko Eckert, Mark Oskin, and Tushar Krishna. Dub: Dynamic underclocking and bypassing in nocs for heterogeneous gpu workloads. In *Proceedings of the 15th IEEE/ACM International Symposium on Networks-on-Chip*, NOCS '21, page 49–54, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Robert A. Bridges, Neena Imam, and Tiffany M. Mintz. Understanding gpu power: A survey of profiling, modeling, and simulation methods. *ACM Comput. Surv.*, 49(3), sep 2016.

[8] 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 M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. 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 García, 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 Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311, 2022.

[9] Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. Effects of dynamic voltage and frequency scaling on a k20 gpu. In *2013 42nd International Conference on Parallel Processing*, pages 826–833, Oct 2013.

[10] Miro Hodak, Masha Gorkovenko, and Ajay Dholakia. Towards power efficiency in deep learning on data center hardware. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1814–1820, 2019.

[11] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018.

[12] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on the gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 221–228, 2010.

[13] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, Shinobu Miwa, and Hiroshi Nakamura. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 349–356, 2013.

[14] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *CoRR*, abs/1910.09700, 2019.

[15] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed DNN training framework with automated 3d parallelism for giant foundation models. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1466–1478, may 2023.

[16] Shigang Li and Torsten Hoefler. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking,*

Storage and Analysis, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Seyed Morteza Nabavinejad, Sherief Reda, and Masoumeh Ebrahimi. Batchsizer: Power-performance trade-off for dnn inference. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ASPDAC '21, page 819–824, New York, NY, USA, 2021. Association for Computing Machinery.

[18] 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.

[19] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7937–7947. PMLR, 18–24 Jul 2021.

[20] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[21] NVIDIA. NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[22] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. The carbon footprint of machine learning training will plateau, then shrink. *Computer*, 55(7):18–28, 2022.

[23] David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *CoRR*, abs/2104.10350, 2021.

[24] 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)*, 2021.

[25] 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*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *CoRR*, abs/1906.02243, 2019.

[27] Zhenheng Tang, Yuxin Wang, Qiang Wang, and Xiaowen Chu. The impact of gpu dvfs on the energy and performance of deep learning: An empirical study. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, e-Energy '19, page 315–325, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Farui Wang, Weizhe Zhang, Shichao Lai, Meng Hao, and Zheng Wang. Dynamic GPU energy optimization for machine learning training workloads. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2022.

[29] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *USENIX NSDI*, 2023.

[30] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

Decentralized Application-Level Adaptive Scheduling for Multi-Instance DNNs on Open Mobile Devices

Hsin-Hsuan Sung[1], Jou-An Chen[1], Wei Niu[2], Jiexiong Guan[2], Bin Ren[2], and Xipeng Shen[1]

[1]Department of Computer Science, North Carolina State University
[2]Department of Computer Science, William & Mary

Abstract

As more apps embrace AI, it is becoming increasingly common that multiple Deep Neural Networks (DNN)-powered apps may run at the same time on a mobile device. This paper explores scheduling in such multi-instance DNN scenarios, on general *open* mobile systems (e.g., common smartphones and tablets). Unlike *closed* systems (e.g., autonomous driving systems) where the set of co-run apps are known beforehand, the user of an *open* mobile system may install or uninstall arbitrary apps at any time, and a centralized solution is subject to adoption barriers. This work proposes the first-known decentralized application-level scheduling mechanism to address the problem. By leveraging the adaptivity of Deep Reinforcement Learning, the solution is shown to make the scheduling of co-run apps converge to a Nash equilibrium point, yielding a good balance of gains among the apps. The solution moreover automatically adapts to the running environment and the underlying OS and hardware. Experiments show that the solution consistently produces significant speedups and energy savings across DNN workloads, hardware configurations, and running scenarios.

1 Introduction

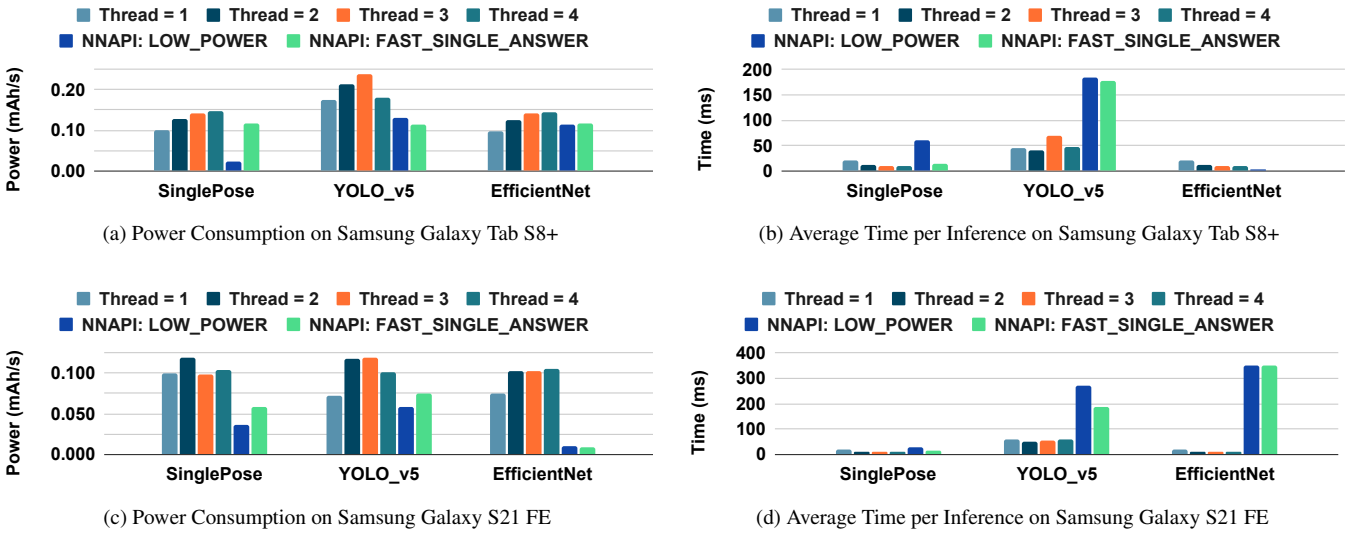
Deep Neural Networks (DNN) have attained remarkable success in various tasks. Recent years have witnessed increasing adoption of DNNs in mobile devices, thanks to the advancement in DNN compression [11, 18, 19], the increasing concerns on privacy, and the demands for real-time responses.

As more apps start to make use of AI, multiple DNN-equipped apps may run on a mobile device at the same time. For example, while a user is using her smartphone to examine some surveillance videos through a DNN-based object detection module, she may be speaking to the DNN-powered personal assistance app on her phone to take notes, while her social media app may be running some DNN-based recommendation algorithm in the background. We call such a co-run scenario *multi-instance DNN* executions.

Scheduling is important for multi-instance DNN executions, especially on resource-constrained systems. This paper particularly focuses on the *spatial* aspect of scheduling, which determines the placement of a DNN-based app on heterogeneous hardware units during each inference. It is critical for the computing efficiency of DNNs. On one hand, DNNs are computationally demanding, and their performance is heavily influenced by the type, configuration, and availability of the underlying computing resources. On the other hand, modern mobile devices (e.g., smartphones, tablets) are commonly equipped with heterogeneous computing units. On a Samsung Galaxy S21, for instance, there is one big "primary" CPU core (ARM Cortex-X1), three medium-sized "performance" CPU cores (Cortex-A78), four small "efficiency" CPU cores (Cortex-A55), one Adreno 660 GPU, and other accelerators. As a result, the speed and power consumption of a DNN running on the different computing units in a mobile device may differ as much as several times as illustrated in Figure 1. Multi-instance DNN executions further complicate the scheduling of DNNs to the best computing units, due to the contentions for computing resources by other co-running DNNs.

The objective of this work is to address such spatial scheduling problems on *open* mobile devices. Here, *open* mobile devices refer to mobile devices on which users can install or uninstall arbitrary apps anytime. In contrast, some devices (e.g., an autonomous driving system) are *closed*, where the applications to install and run are predetermined. The problem of multi-instance DNN scheduling also exists on *closed* devices, and has been explored in some previous studies [4, 9, 14, 28]. But those studies assume that the set of co-running apps are known beforehand and their schedules are fully controllable by a central agent (e.g., OS), which is not the case for *open* devices. Their solutions hence cannot apply to the *open* devices. To the best of our knowledge, no prior solutions have been proposed for multi-instance DNNs scheduling on open mobile systems.

This work proposes the first-known *decentralized* application-level adaptive spatial scheduler for multi-instance DNNs on open mobile devices. Being decentralized means



(a) Power Consumption on Samsung Galaxy Tab S8+

(b) Average Time per Inference on Samsung Galaxy Tab S8+

(c) Power Consumption on Samsung Galaxy S21 FE

(d) Average Time per Inference on Samsung Galaxy S21 FE

Figure 1: **Standalone Profiling of Three DNN Networks.** We profile the average power and inference time for different DNNs. They have their own best option for delegation. Thread = *i* means the model is executed on the CPU with *i* threads. NNAPI *low power* and *fast single answer* are two options that consider using GPU and accelerators.

that the spatial scheduling decisions are made by each application rather than by a centralized agent (e.g., OS). This distinctive design brings several benefits over centralized schemes: It is easy to adopt without the need for OS modifications; it requires no OS admin privileges; it preserves privacy and avoids inter-app communications or app-OS special communications and the associated overhead.

Specifically, the spatial scheduling considered in this work includes the decisions on running a DNN on GPU or on CPU and if on CPU how many CPU threads to use. We intentionally leave the temporal aspect of scheduling (i.e., at what time an app runs or gets evicted) and priority management as they are, because these tasks are what the OS is already taking care of. We meanwhile ensure that the spatial scheduling method can automatically adapt the scheduling decisions to the temporal scheduling by the underlying OS. This design makes the solution easy to adopt (as no OS modifications are necessary), applicable across systems, and workable regardless of what the other co-running apps are and what scheduling policies they follow. It also retains the fairness guarantees and starvation-avoidance provided by the underlying OS.

Our solution achieves these properties by leveraging the adaptivity of Deep Reinforcement Learning (DRL) [17,20,25]. We develop a DRL-based scheduling library. It is decentralized, working at the application level. Any app may call the library to dynamically determine, for the next DNN inference, whether CPU or accelerators is to be used, what modes (e.g., performance or power-efficiency modes) to use, and if CPU, what is the best number of threads to launch. It requires no direct knowledge about other apps. It gives recommendations based on the current state of the executing environment and the estimated rewards this application is expected to obtain for each of the possible schedules—produced by a model learned through the self semi-supervised approach of DRL.

By drawing on previous theoretical results on the Nash equilibrium of RL, we provide discussions on the convergence of the scheduling algorithm and the empirical evidences.

In a set of co-run scenarios formed by the subsets of nine DNNs, the proposed solution improves the average latency by as much as 4×, and saves the average energy consumption by as much as 3× compared to Android NNAPI, the official Android tool that automatically selects the computing units to use for running a DNN. Experiments on a smartphone (Samsung Galaxy S21) and a tablet (Samsung Tab S8+) show that the benefits are consistently significant. Further experiments show that the benefits remain even if those DNNs co-run with uncontrolled apps (i.e., apps that do not employ the proposed scheduling algorithm). The scheduling algorithm converges quickly (within seconds) and adapts to the running environments automatically, making it an immediately adoptable solution across mobile systems.

Overall this work makes the following main contributions:

- To our best knowledge, the solution in this work is the first decentralized application-level adaptive scheduling for multi-instance DNNs on open mobile devices.
- This work uncovers a set of novel insights: (i) It is possible for a scheduler to work effectively without direct knowledge of other apps in multi-instance DNN scheduling; (ii) the DRL-based scheduling is effective in adapting to the factors in the execution environment (OS, other apps, priorities, etc.); (iii) the decentralized scheduling

algorithm is quick in converging to a balance point.

- This work empirically evaluates the efficacy of the proposed solution, showing that it consistently gives significant speedups and energy savings across DNN workloads, hardware configurations, and running scenarios (with or without uncontrolled apps, various background/foreground combinations).

2 Background

Reinforcement Learning and Deep-Q-Network Reinforcement learning (RL) [20] is a machine learning approach in which an agent learns from environmental feedback and a series of decisions to maximize the total cumulative rewards to stimulate better decision-making. Q-learning [27] is a type of RL algorithm that seeks an offline policy to maximize the expected total rewards across all steps. "Q" refers to the policy function $Q(S,A)$ that inputs a state and action pair and outputs a Q-value. Two typical Q-learning schemes are Q-table and Deep-Q-network (DQN) [17]. DQN improves the limited representation of the Q-table.

Q-table stores the state and action pairs with the estimated Q-value. The Q-value of each step can be obtained by table lookup. However, this approach only works when states and actions are discrete values and the sets are small. In contrast, DQN replaces the Q-table with a neural network (NN) as a function approximator. Now, the sets of states and actions can be large, and the state space can be continuous. In the training phase, the loss function minimizes the squared error between the target Q-value and the predicted Q-value given by the NN. So the loss function of NN can thus be formulated as follows: $L = (Q(S,A) - (R + \gamma * \max(Q(S',A'))))^2$, where R is the immediate reward for taking action A in state S, γ is the discount factor (a value between 0 and 1 that determines the importance of future rewards), and S' and A' are the next state and the possible actions in that state, respectively.

Nash Equilibrium The Nash equilibrium is a concept in game theory that describes the optimal behavior of players in a game. It is a stable, self-enforcing, and Pareto optimal solution, where each player has chosen an optimal action, given the other players' actions. There are several methods for finding the Nash equilibrium of a game, including using best response functions and mixed strategies. The Nash equilibrium is a helpful tool for analyzing strategic interactions and predicting players' behavior in a game.

One way to find the Nash equilibrium is to use the best response function, which maps each player's action to the action that maximizes their reward, given the actions of the other players. The Nash equilibrium is then the set of actions where each player's action is the best response to the actions of the other players. Another approach is to use a mixed strategy, where players randomly choose their action with a certain probability. The Nash equilibrium is then the set of possibilities where each player's mixed strategy is the best response to the combined strategies of the other players.

3 Problem Statement and Research Questions

This section first provides a definition of the focused scheduling problem, and then lists the important research questions.

3.1 Problem Definition

Given: A set of apps A that may execute on a device V, and some of them may run at the same time. $A = A_c \cup A_u$, where, each app in A_c contains some DNNs and employs a policy P to decide the execution configuration of each of its DNNs, while the apps in A_u do not follow policy P. For each DNN, there are K possible configurations which affect the usage of CPUs and GPUs of the DNN differently.

Objective: Finding P such that the following is minimized:

$$\sum_{i \in A_c(DNN)} \sum_{j} L_{i,j} * W_{i,j}$$

where, $L_{i,j}$ and $W_{i,j}$ are respectively the latency and power consumption of the j^{th} inference of DNN_i, $A_c(DNN)$ is the DNNs in the apps in A_c. We use the product of latency and power to capture the common interest in both speed and energy usage on mobile devices.

Constraints: (1) The scheduling policy P in an app cannot access the information of another app (due to the isolation enforced by mobile systems); (2) each app's priority and temporal scheduling are controlled by the underlying OS.

3.2 Design Considerations and Principles

The main aspect of scheduling focused in this work is the execution configuration that affects the usage of computing units—which is essential for the performance and power consumption of DNN. The relevant factors include some that are controllable at the application level, and some at the OS level. For a DNN written in TFLite [3] (a popular development framework for mobile AI), for instance, the application can explicitly specify whether the DNN should run on CPU or accelerators (e.g., GPU), and the number of CPU threads to use. It may also call APIs in NNAPI [8] (the Android official library for running DNNs) with either a performance mode or an efficiency mode; the APIs will automatically determine the CPUs or accelerators to be used for the DNN. We uniformly regard such configurations as application-level controllable configurations, which specifically include the explicit specification of computing units and CPU thread number and the calls to other relevant libraries.

When a DNN runs with a certain configuration, the OS may exert further influence on the usage of the computing units.

For instance, if the application-level control sets the DNN to run on a CPU with 2 CPU threads, the OS will ultimately determine which two CPU cores the threads will run on. If they are small cores, the speed may be much lower than on medium cores.

We design our solution with the following principles:

(1) The scheduling should be decentralized, functioning inside each application. This is because most of the relevant factors are inside the app, and application-level solutions are easier to adopt as they do not need changes to the underlying OS.

(2) The scheduling should be able to adapt to the influence of the underlying OS and other environmental factors (e.g., priorities of apps, background/foreground differences).

(3) The scheduler should work efficiently and adapt to changes in the system agilely.

3.3 Research Questions

Creating a decentralized application-level scheduler on open mobile systems has many differences from centralized scheduling on closed systems and hence raises many new questions. We summarize them into the following six open research questions (RQ). They are gradually addressed in the rest of this paper.

RQ1: How can the scheduler work effectively without direct knowledge of other apps?

The apps on a smartphone may be developed by many different authors. For security and privacy, open mobile systems typically impose strong isolations among apps. One app cannot access the direct info of another app. How can the scheduler work well under such a constraint?

RQ2: How can the solution deal with the effects of the scheduler in the underlying OS?

OS influences the execution of the Apps: Ultimately, it is the OS that allocates computing units and other resources to each App and determines when an app runs and its priority level. The OS schedulers differ from one version of OS to another. We avoid demanding changes to the underlying OS for easy adoption of our solution. The user-level scheduling hence must be made adaptive to OS.

RQ3: Can decentralized scheduling converge to a good result?

On an open mobile system, apps come and go, and the workloads on the system may vary continuously. Without the knowledge of other co-running apps, can decentralized scheduling converge to good results?

RQ4: How fast can the decentralized scheduling learn and adapt?

Machine learning-based decision models usually take time to learn, but the dynamic nature of mobile systems demands fast responses. Can decentralized scheduling meet the speed needs?

RQ5: Can the solution work if there are uncontrollable apps?

In real mobile usage scenarios, not all Apps will adopt the same scheduling policy. The workload from uncontrollable apps can be unpredictable. Can the proposed solution still function well in the presence of such uncontrollable apps?

4 Decentralized DQN Scheduler

Algorithm 1: DQN Algorithm for each Apps

Input: Environment E; Replay Memory M; Exploration Ratio ε; The parameters of Policy Network θ and Target Network θ^-; Discount Factor γ; Batch Size B; Update Steps C; Huber loss function L

Output: $Q(s,a;\theta)$

Initialize: Take observation from E and generate current state s

```
while Inference start do
    if rand() < ε then
        Select action a randomly
    else
        Select action a ← argmax_a Q(s,a; θ)
    Run inference on a target defined by action a
    When Inference ends, Calculate reward r
    Observe from E and generate next state s'
    Store transition (s, a, s', r) to M
    if M.size() > B then
        Sample a mini-batch N from M
        for each transition (s_j, a_j, s'_j, r_j) in N do
            y_j = r_j + γ max_a' Q(s_j, a' : θ^-)
            Calculate Loss l_j = L(y_j, Q(s_j, a_j : θ))
        Batch Update θ using SGD algorithm by loss vector l
    s ← s'
    i ← i + 1
    if i mod C == 0 then
        θ ← θ^-
```

In this section, we introduce the design of the decentralized scheduler, which also answers RQ1 and RQ2.

The design is based on deep reinforcement learning (DQN). As Section 2 mentions, DQN is a deep reinforcement-learning method. As a semi-supervised method, it requires no manual labels, but actively explores the environment, learns the relations between actions and rewards automatically, and uses the learned model to predict the next suitable action. This nature makes it a good fit for the dynamic environment in our problem. In contrast, a DNN-based approach would require offline labels of many training cases and be slow in adapting to dynamic changes.

The DQN-based RL agent is also light-weighted and converges fast. For the storage overhead, the total extra mem-

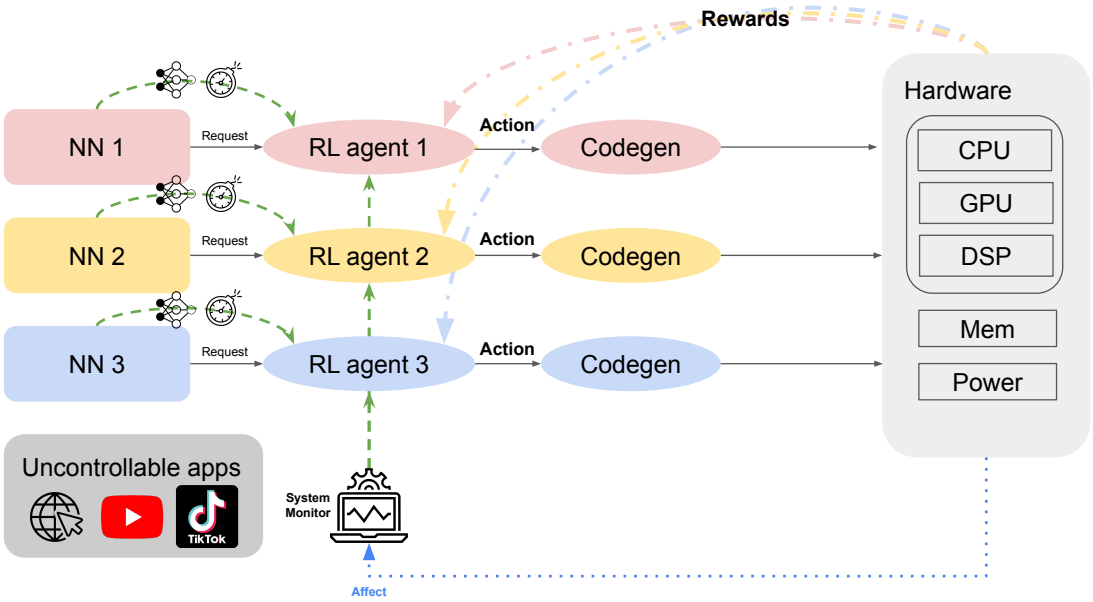


Figure 2: **The Structure of Decentralized DQN Scheduler.** The figure shows three controllable DNN applications that have their RL agent that selects actions (different code generation of the model). All of them collect system status as their RL agent input state. Also, we include the co-running uncontrollable apps that do not contain RL agents inside.

ory footprint needed less than 250KB. For the computation overhead, the inference time of the network only takes less than 1ms and one training process only takes about 1ms to 2ms when three DQN agents co-running. In general real-time constraints are 50ms for each inference, and those extra computations overhead in each model can be neglected.

Figure 2 illustrates the role of the scheduler in a multi-instance scenario. Other than the uncontrolled apps, each DNN-based app contains a DQN agent for configurations. They do not have access to other apps but can obtain the state of the resource utilization of the whole system.

We define the learning procedure of our DQN-based agents with *States*, *Rewards*, and *Actions* in the multi-DNN execution environment as follows:

States We use the following variable to capture the static and variance environment information. For static features, we use the number of convolution layers, the number of fully-connected layers, and the MAC operations of DNN models. For environment variance information, we use the CPU utilization, GPU utilization, and memory usage of co-run apps.

Rewards The reward function R is composed of three important metrics: latency, power consumption, and deadline. It is defined as follows:

$$R(L_i^{s,a}, W_i^a, d_i) = \begin{cases} -1000 & \text{, if } \quad L_i^{s,a} > d_i \\ -L_i^{s,a} \times Power_i^a & \text{, otherwise} \end{cases}$$

where, $L_i^{s,a}$ represents the latency of the inference of DNN_i with action a on state s, W_i^a is the average power of the inference of DNN_i with action a, and d_i is the deadline for the inference of DNN_i.

Because ($latency \times power$) can be regarded as the energy consumption estimation, we call it **Energy Factor** in our paper. Also, it is used as one of the metrics in our evaluation (Section 6). The multiplication of power and latency gives a linear relationship, with no skew towards either factor. For instance, if the latency doubles, the entire reward function doubles, and the same applies to power consumption. The simple production form of the reward function avoids additional hyper-parameters. A large negative reward is used when a deadline is passed to discourage missing deadlines.

Actions The action space involves the configurations that can affect the usage of computing resources on the hardware. In our study, we include six configurations as detailed in Section 6.1.

Neural Networks DQN includes a *policy neural network* and a *target neural network* inside, which learn about the relations between states, actions and rewards. We want to make the networks as simple as possible to control the runtime overhead. So we adopt a model that only maintains two fully-connected (FC) layers as our policy and target network. The first FC layer input channel is 8, and the output channel is 100. The second FC layer input channel is 100, and the output channel is 6. For the input, it is the vector of the state. Its dimension is 8, which contains four CPU and GPU usage pairs. The output is a vector that represents the q-values of six actions.

The DQN agents go through an exploration and learning

stage until they reach convergence. As listed in Algorithm 1, at the start of each episode, each agent selects the action with the maximum q-value generated by the policy network with parameters θ and current state s but has some exploration rate ε to select action randomly. Here, one episode is one inference of the DNN model. After making the inference with the selected action, the power consumption and latency are collected to compute the reward r. Then, the agent observes the environment and generates the updated state s'. One transition (s, a, s', r) is then pushed into the replay memory M. The training process starts when the replay memory M has enough data. It goes as follows. It first samples a mini-batch N from M. Then, it calls the *target network* to generate the expected q-value y for each transition. After that, it uses the Huber Loss function [10] to calculate the loss value between the expected q-value and the current q-value output by the *policy network*. The final step of the training process is to update the parameters θ based on the loss value. In every C episode, the parameters θ of the *policy network* are copied to the *target network* as its new parameters. In the targeted co-run scenario, each controllable App is equipped with a DQN agent that is trained for scheduling its DNN inferences.

5 Convergence Discussion

This section discusses the convergence of the DQN-based scheduling algorithm (RQ3). Prior works [6] generalize the multi-agent Q-learning method as a general-sum stochastic game and prove all reward functions in each agent are guaranteed to converge. Specifically, Hu and Wellman [13] present an algorithm to solve the general-sum stochastic games, and Bowling [6] strengthens the proof with further assumptions.

Their convergence theorem has four necessary assumptions, two of which are about exploration and the decay of the learning rate, and are similar to those used in the Deep-Q-Learning algorithm. It is assumed that they have been met. The remaining two assumptions [6, 13] are as follows:

Assumption .1 *A Nash equilibrium* $(\pi^1_*(s), \pi^2_*(s))$ *for any stochastic game* (Q^1_n, Q^2_n) *satisfies one of the following properties:*

1. The equilibrium is a global optimal.

2. The equilibrium receives a higher payoff if the other agent deviates from the equilibrium strategy.

Assumption .2 *The Nash equilibrium of all stochastic games* $Q_n(s)$*, as well as* $Q_*(s)$ *must satisfy property 1 in Assumption 1* ***or*** *the Nash equilibrium of all stochastic games,* $Q_n(s)$*, as well as* $Q_*(s)$ *must satisfy property 2 of Assumption 1*

Assumption 1 includes a property that states that there exists a set of strategies for the agents, where each agent individually obtains the highest possible payoff. This also guarantees that this set of strategies forms an equilibrium since no agents would gain from deviating from their chosen strategy. Assumption 2 includes another property in which the game's Nash equilibrium is a "saddle point." This implies that if an agent deviates from the equilibrium, the agent would not gain, but other agents would, which makes no agent want to deviate from the equilibrium.

Finding a globally optimal solution for the multi-agent problem is known to be NP-hard [7]. However, by reaching Nash equilibrium during convergence, RL can ensure that each agent adheres to a strategy that gives a good payoff to both itself and the other agents. Reflected in our scheduling context, it means that all controllable Apps may adhere to a strategy that helps meet their deadlines while minimizing energy consumption. The achievement of Nash equilibrium eliminates fairness concerns among controllable Apps, as they all reach a stable state where each maximizes its benefits within the given constraints.

Based on prior studies [6, 13], it has been demonstrated that a zero-sum stochastic game converges. Our scheduling problem for multi-DNN applications in this paper bears a resemblance to a zero-sum stochastic game. In our case, each agent corresponds to our DQN agent for each controllable DNN application. All DQN agents involved compete for limited resources, such as CPU and GPU, with a maximum utilization boundary. While our problem may not strictly adhere to all the definitions of a stochastic game, we empirically demonstrate its convergence under our circumstances in Section 6.

6 Evaluation

This section evaluates our decentralized application-level adaptive scheduler (called DQN for short in this section) by comparing it with three baseline scheduling methods that are designed for single DNN execution: two static scheduling settings used by Android Neural Networks API (NNAPI) [8] (`NNAPI LOWER_POWER` that minimizes the power consumption for each DNN and `NNAPI FAST_SINGLE_ANSWER` that minimizes the inference latency for each DNN) and an offline profiling-based scheduling method (`Best Standalone` that based on offline profiling selects the best setting for each individual DNN among all delegate settings introduced in Section 6.1). This evaluation has three objectives as follows: 1) demonstrating that as the *first* decentralized DNN co-run scheduling method, DQN outperforms all baseline scheduling approaches that are designed for single DNN execution in multiple representative DNN co-run scenarios (Section 6.2); 2) verifying DQN's convergence and fast converging speed, and studying the underlying reason why DQN outperforms baseline scheduling methods by a reward convergence analysis (Section 6.3); 3) proving DQN's benefits remain even if the DNNs co-run with uncontrolled apps with both predictable and unpredictable workloads (Section 6.4).

6.1 Evaluation Methodology

Benchmarks. Table 1 characterizes the nine DNN models from various domains used in the evaluation[1]. All models are in TensorFlow Lite [3] format. These DNN models form three groups: `Image`, `Audio & Image`, and `Video & Image` with three models in each group. Our evaluation runs models in each group simultaneously to simulate the real-world DNN co-run scenario. For example, Group 1 (G1) simulates an intelligent camera running varied AI capabilities simultaneously, pose detection (`SinglePose`), object detection (`YOLO-v5`), and image classification (`EfficientNet`). Other groups simulate more complex scenarios with audio and image or video and image co-processing. In addition, these DNNs have varied model sizes, standalone latency, and power consumption, representing three different cases: relatively balanced workloads, mild imbalanced workloads, and severe imbalanced workloads, respectively. Therefore, they show different behaviors in the evaluation. Please find more discussions in Section 6.2.

Table 1: **Nine DNN Models Used in Our Evaluation.** They form three groups. DNN models in each group are executed simultaneously.

Group	Models	Sizes (KB)
G1: Image	SinglePose [1] (SP)	9,154
	YOLO-v5 [5]	7,428
	EfficientNet [23] (ENet)	6,265
G2: Audio&Image	YamNet [24]	4,031
	MobileNetv1 [12] (MNv1)	4,188
	WDSR [29]	1,252
G3: Video&Image	Movenet [16]	24,440
	Esrgan [26]	4,877
	MobileNetv2 [21] (MNv2)	13,666

Table 2: **Absolute Latency of DQN.** This table reports the absolute latency of DQN in Figure 3, Figure 6, and Figure 7.

	Figure 3		Figure 6		Figure 7	
(Uint: ms)	Tablet	Phone	Tablet	Phone	Tablet	Phone
G1: SP	8.5	10.7	35.0	32.5	32.1	34.4
G1: YOLOv5	331.1	397.1	591.9	318.7	542.4	446.6
G1: ENet	5.0	47.2	9.7	337.8	7.0	341.3
G2: YamNet	4.3	3.8	23.4	23.0	20.4	17.1
G2: MNv1	6.2	34.3	10.5	30.8	8.2	32.1
G2: WDSR	6.8	116.9	6.3	114.2	5.0	108.2
G3: Movenet	34.2	33.0	76.2	71.3	78.5	70.1
G3: Esrgan	62.3	61.5	64.6	75.2	94.4	69.8
G3: MNv2	40.2	25.2	80.2	42.5	17.1	35.4

Software settings.

Our evaluation considers two co-run scenarios: *controllable DNN tasks co-run* and *uncontrollable tasks co-run. Controllable DNN tasks* refer to Apps that incorporate our RL agents, while *uncontrollable tasks* refer to Apps that do not involve our RL agent. *Uncontrollable tasks* may or may not use DNNs.

[1]These DNNs are collected from Tensorflow Hub [2] and GitHubs [22].

Table 3: **Absolute Energy Factor of DQN.** This table reports the absolute energy of DQN in Figure 3, Figure 6, Figure 7.

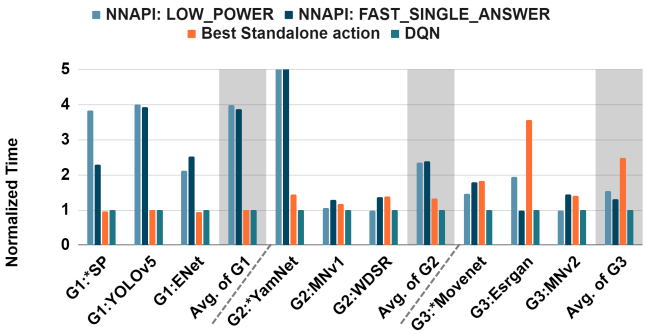
	Figure 3		Figure 6		Figure 7	
(Uint: Joule)	Tablet	Phone	Tablet	Phone	Tablet	Phone
G1: SP	18.9	15.9	13.48	18.1	25.2	21.8
G1: YOLOv5	950.3	446.3	1.6k	365.1	1.5k	509.8
G1: ENet	9.8	122.3	16.8	67.9	13.8	69.6
G2: YamNet	7.7	7.0	17.4	39.9	52.5	33.5
G2: MNv1	11.3	58.9	12.5	57.4	19.8	64.9
G2: WDSR	11.0	41.7	10.7	58.2	11.2	35.3
G3: Movenet	48.0	19.3	73.3	26.6	59.3	29.5
G3: Esrgan	33.2	14.5	36.0	20.4	23.4	16.4
G3: MNv2	27.7	32.8	11.8	38.2	60.8	29.6

Controllable DNN Tasks Co-Run. We build an Android demo app with Java that can run each DNN individually. Users can control this demo app to start and stop DNN inference. This demo app relies on TensorFlow Lite (TFLite) [3] to run DNNs. Our evaluation employs multiple delegate settings in TFLite to run DNNs: using 1, 2, 3, or 4 CPU threads[2], respectively, and using two NNAPI [8] modes, LOWER_POWER or FAST_SINGLE_ANSWER, respectively. Particularly, NNAPI is designed for accelerating TensorFlow Lite DNN execution on mobile devices with supported hardware accelerators including GPU, DSP, and NPU. It automatically partitions a DNN model, maps each partition to a processor, and calls corresponding kernel codes for that processor. Thus, we can treat it as *static offline scheduling* for each individual DNN. Our evaluation particularly employs two NNAPI modes as the baseline, `LOWER_POWER` which minimizes the power usage, and `FAST_SINGLE_ANSWER` which minimizes the inference latency. Besides them, our evaluation also employs an offline profiling-based scheduling method as a baseline: `Best Standalone` that selects the best setting (i.e., with the best energy factor defined as $power\_consumption \times latency$) for each individual DNN among all delegate settings (including using 1 to 4 CPU threads and two NNAPI modes) based on offline profiling results[3].

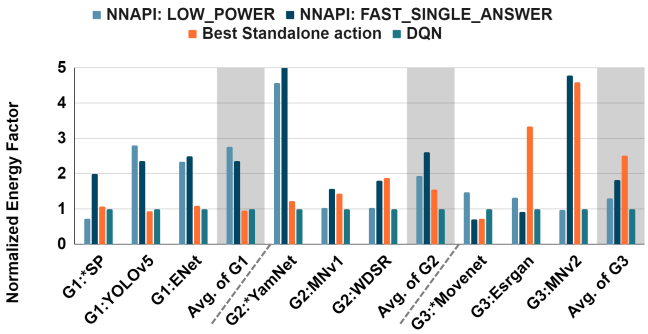
Uncontrollable Tasks Co-Run. We select two widely used real-world applications `TikTok` and `Web Browser` to experiment with two popular user behaviors, watching social media video and browsing web pages. Here we select the default web browser in Android, Google Chrome as our target.

[2]The evaluated mobile chip has 8 CPU cores, but the Android OS only allows background Apps to access the 4 small CPU cores. Thus, we make it consistent throughout our evaluations: the foregrounds Apps access the 4 cores (prioritize to access the big core, medium core, then small cores), and the background Apps access the 4 small cores.

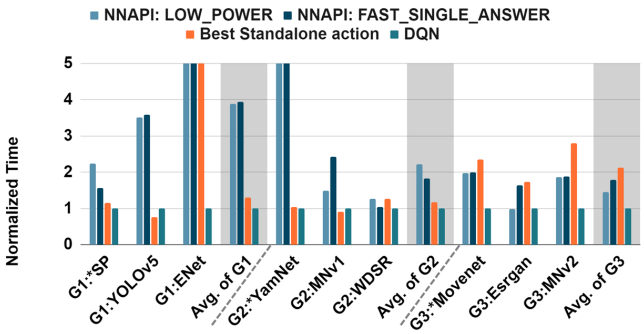
[3]we profile the power values through the Android Developer API "dumpsys batterystats".



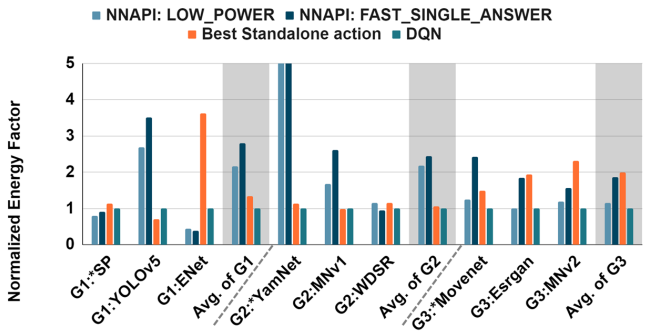
(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+



(b) Normalized Energy Factor on Samsung Galaxy Tab S8+



(c) Normalized Average Inference Time on Samsung Galaxy S21 FE



(d) Normalized Energy Factor on Samsung Galaxy S21 FE

Figure 3: **Overall Comparison between DQN and Three Baselines on Three DNN Groups.** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each group co-running three DNN apps (* denotes this DNN runs in the foreground and the others run in the background).

Evaluation platforms. DQN is evaluated on two edge devices: (1) `Samsung Galaxy S21 FE 5G` mobile phone, equipped with Android 12 OS, and Qualcomm SM8350 Snapdragon 888 5G SoC with Octa-core CPU (1x2.84 GHz Cortex-X1 & 3x2.42 GHz Cortex-A78 & 4x1.80 GHz Cortex-A55), Adreno 660 GPU (Version 1), and Hexagon 780 DSP. Its storage capacity is 128GB with 6GB RAM and its voltage is 4.3V. (2) `Samsung Tab S8+ tablet`, equipped with Android 12 OS as well, and Qualcomm SM8450 Snapdragon 8 Gen 1 SoC with Octa-core (1x3.00 GHz Cortex-X2 & 3x2.50 GHz Cortex-A710 & 4x1.80 GHz Cortex-A510), Adreno 730 GPU, and Hexagon DSP. Its storage capacity is 128GB with 8GB RAM and its voltage is 4.1V.

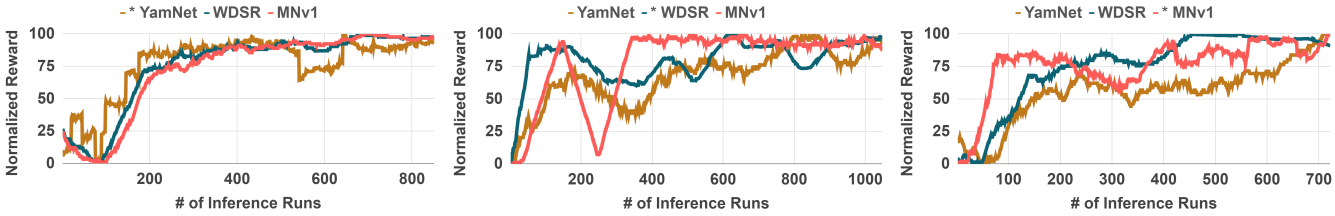
6.2 Overall DQN Scheduling Performance

This section evaluates DQN on the three groups of co-run DNNs in Table 1 by comparing it with the three scheduling baselines aforementioned: `NNAPI LOWER_POWER`, `NNAPI FAST_SINGLE_ANSWER`, and `Best Standalone`. Figure 3 shows the comparison results, in which the x-axis shows the three DNNs in each group and the average performance of each group. It is worth noting that the *star (*)* before the DNN name indicates that this DNN model is executed in the foreground (and two other DNNs in the same group are executed in the background) for this co-run[4]. We intentionally use this setting to simulate the real-world Apps co-run on the Android system (and bring the OS impact on the user-level scheduling into account).

Figure 3 employs two metrics to compare our decentralized DQN system with three baselines: `average inference latency` (as shown in Figure 3a and Figure 3c) and `energy factor` (as shown in Figure 3b and Figure 3d). The energy factor is defined as $power\_consumption \times latency$ for each inference, which is also used as our reward function in each DQN agent, the lower the better. To improve the readability, we normalize the results in Figure 3 by setting DQN performance as 1. Table 2 and 3 summarizes the absolute values for reference.

Figure 3 shows that for the average inference latency, our decentralized DQN-based scheduler achieves up to 4× speedup over two baselines of NNAPI (`NNAPI LOWER_POWER` and `NNAPI FAST_SINGLE_ANSWER`), and 2.7× speedup over

[4]Android OS grants foreground and background Apps different priorities/limitations, e.g., normally, background Apps have lower priority than foreground ones, and background Apps cannot access the big core of CPUs.



(a) YamNet at Foreground (b) WDSR at Foreground (c) MNv1 at Foreground

Figure 4: **Reward Convergence Trend for the Three Apps Co-Run in G2.** It reports DQN's normalized reward trend to prove DQN converges in different situations. It uses DNNs in G2 and places each DNN in the foreground.

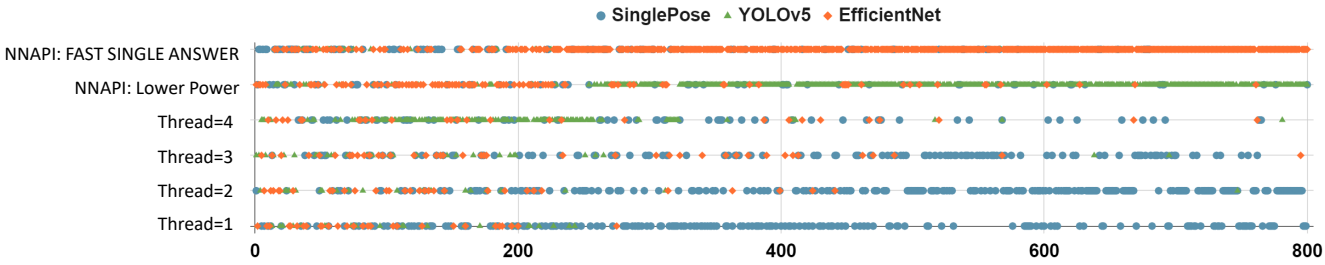


Figure 5: **Selected Action Convergence Trend.** This figure shows the action selection through runs for co-running results in G1 (all of them are in the background). The action selection will be converged into one or two options.

Table 4: **Selection Rates of Actions of Last 100 Runs of Figure 5.**

	Thread = 1	Thread = 2	Thread = 3	Thread = 4;	NNAPI: LOW_POWER	NNAPI: FAST_SINGLE_ANSWER
SinglePose	14%	75%	6%	1%	1%	3%
YOLO_v5	0%	1%	1%	0%	98%	0%
EfficientNet	0%	3%	0%	3%	0%	94%

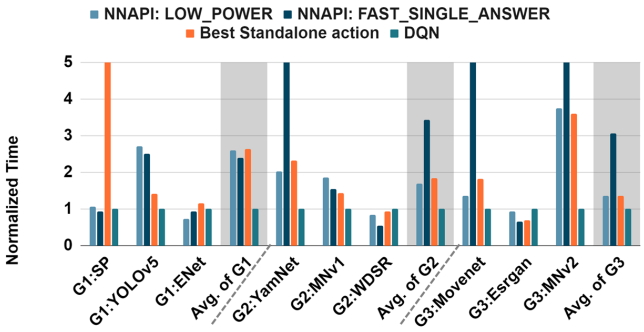
`Standalone Best` action selection, respectively, for average results of three co-running DNN groups (gray area). For the energy factor, our decentralized DQN-based scheduler achieves up to 3× energy saving over `NNAPI LOWER_POWER` and `NNAPI FAST_SINGLE_ANSWER`, and 2.6× energy saving over `Standalone Best` action selection, respectively, for average results of three co-run DNN groups. Comparing the DQN performance across three groups, we can see that as the workload imbalance increases (from G1 to G3), the benefit of DQN over `Best Standalone` grows while its benefit over both NNAPI schedulers drops. This is mainly because `Best Standalone` partitions each DNN workload into more processing units than NNAPI schedulers, so it reduces the resource competition caused by the workload imbalance. DQN has a similar effect. Moreover, although the average inference latency and energy factor vary for different groups (and each DNN model) under various settings, our decentralized DQN-based scheduler always performs better than baseline methods. These results prove that DQN is robust enough to deliver high-quality scheduling results for various DNN applications and environment settings (e.g., varied foreground/background DNN settings, DNN structures/target domains, and executing devices). The following sections further verify this claim.

6.3 Reward Convergence Analysis

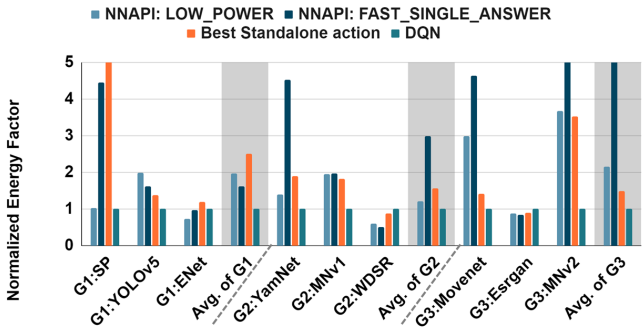
This reward convergence analysis has three objectives: 1) to verify the rewards of multiple agents converging empirically, 2) to measure the DQN convergence speed[5], and 3) to study why DQN outperforms baseline scheduling by analyzing its trend of action selection.

We verify the convergence and measure the convergence speed by taking Group 2 (in Table 1) as an example due to the space limitation and other groups show similar trends. Figure 4 shows the convergence trends of DQN training on co-running three DNNs in Group 1 under three different set-

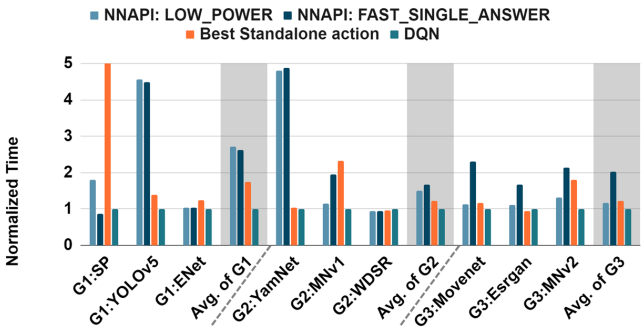
[5]Theoretical convergence proof only proves all DQN converges eventually without showing the convergence speed.



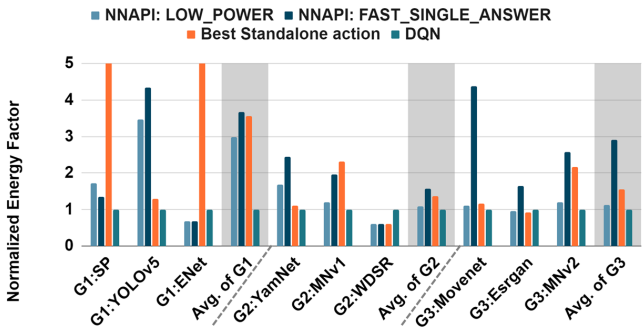
(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+



(b) Normalized Energy Factor on Samsung Galaxy Tab S8+



(c) Normalized Average Inference Time on Samsung Galaxy S21 FE



(d) Normalized Energy Factor on Samsung Galaxy S21 FE

Figure 6: **Co-Run Three DNN Groups with Uncontrollable App TikTok.** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each DNNs group co-running with uncontrollable app TikTok that plays video posts on the foreground.

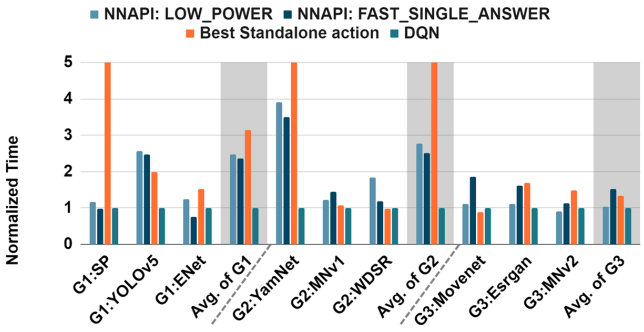
tings that place `YamNet`, `WDSR`, or `MNv1` at the foreground, respectively. The x-axis is the number of DQN inference runs (which is equivalent to the number of DNN inference runs), and the y-axis is the average action selection rewards of 100 times. Regardless of the environment settings, the DQN agents can reach convergence within 800-1000 inference runs. The scheduling time overhead for a single execution of the DNN App includes the forward- and backward-propagation phases of the DQN agent, which typically take 1.2 ms on average and only contribute to 0.5-5% of the overall DNN App execution time (which ranges 20-176ms). Additionally, the energy overhead amounts to approximately 0.5-2% of the DNN App execution. This convergence time is trivial compared to the time required to profile and configure each model manually.

We next study why the DQN agent can outperform the single DNN scheduling baselines by analyzing its trend of action selections. We take Group 1 (in Table 1) with `SinglePose`, `YOLOv5`, and `EfficientNet` as an example this time. All models run in the background, and the evaluation results are shown in Figure 5. In addition, Table 4 shows the percentage of action selection in the last one hundred runs to give an insight into action selection after the model is converged.

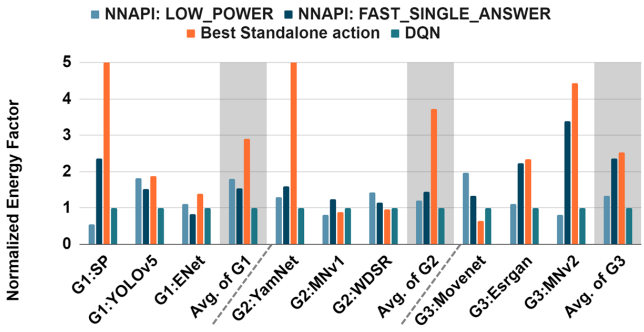
Figure 5 and Table 4 offer us two key insights in DQN: first, even though each DNN starts with multiple selections, their decision is converged into one or two actions, and second, a precise boundary exists between the actions only using CPU and those cooperating with GPU. The regular pattern of the action selection in the first insight implies that the DQN model converges at the end, empirically proving that the multi-agent DQN game converges (in another setting). The second insight tells us that DQN can effectively avoid computing resource contention. For example, when YOLOv5 and EfficientNet run on GPUs, DQN is able to schedule SinglePose on CPUs, thus preventing competition for the limited resource of accelerators; while other baseline scheduling methods fail to do this, resulting in GPU contention.

6.4 DQN Performance w/ Uncontrollable Apps

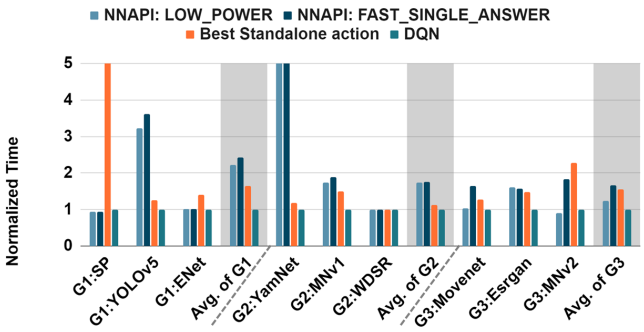
This section evaluates the performance of our decentralized DQN scheduler under uncontrollable application co-running situations. More specifically, we aim to 1) compare the performance between our decentralized DQN scheduler and those



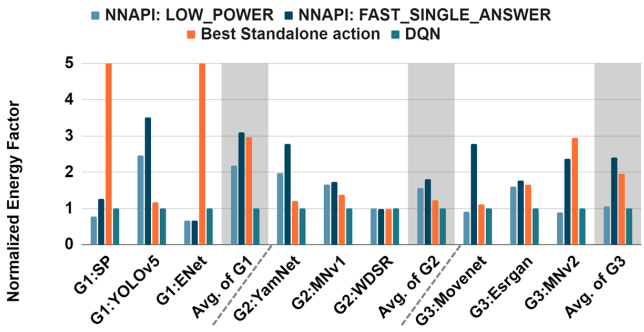
(a) Normalized Average Inference Time on Samsung Galaxy Tab S8+



(b) Normalized Energy Factor on Samsung Galaxy Tab S8+



(c) Normalized Average Inference Time on Samsung Galaxy S21 FE



(d) Normalized Energy Factor on Samsung Galaxy S21 FE

Figure 7: **Co-Run Three DNN Groups with Uncontrollable App Web Browser (Google Chrome).** Compare (a) normalized average inference time and (b) rewards on Samsung Galaxy Tab S8+ and the same metrics (c) and (d) on Samsung Galaxy S21 FE when each DNNs group co-running with the uncontrollable app Google Chrome randomly browsing pages on the foreground.

baselines under the popular real-world Apps co-running circumstance, and 2) illustrate our DQN solution has strong adaptivity when co-running with Apps with dynamically changing workloads.

Specifically, we simulate the real-world environment by simultaneously running DNNs and other apps with predictable workloads (i.e., repeatedly playing a TikTok video post), and unpredictable workloads (i.e., randomly browsing webpages in a web browser, Google Chrome). Both TikTok and Google Chrome run in the foreground, while DNNs (and their demo applications) run in the background. TikTok and Google Chrome require CPU and accelerators (e.g., GPUs), thus careful workload scheduling of DNNs is desired if we would like to achieve optimized system performance. To verify the predictability of TikTok and the unpredictability of Google Chrome, respectively, we use `GPUWatch` to monitor both applications' execution and find that the major task, video processing in Tiktok requires a stable amount of GPU resources, while Google Chrome only consumes GPU resources when users touch or swipe across the screen, resulting in irregular GPU usage.

Figure 6 and Figure 7 compare DQN with all three baselines aforementioned, `NNAPI LOWER_POWER`, `NNAPI FAST_SINGLE_ANSWER`, and `Best Standalone` on two platforms under two uncontrollable cases (more predictable TickTok and more unpredictable Google Chrome), respectively. Our decentralized DQN-based approach outperforms all baselines for both cases in terms of both latency and energy for all three groups of DNNs. For the Tiktok case, DQN shows better performance because the DQN agent has more convincing history data that can predict more accurate action for the next step. The Google Chrome case empirically proves that our decentralized DQN-based approach is robust enough to handle DNNs co-run with an app that has unpredictable workloads.

Compare with a Heuristic-Based Adaptive Method. To confirm that simple heuristic-based adaptation is insufficient for the scheduling problem, we implement a heuristic method called "trial and set" (T&S). In T&S, each action performs X (50 in our experiments) inference runs and selects the action with the minimum observed online energy factor in subsequent inferences. We compare it with our DQN results on G2 in each of three settings: three DNNs in G2 co-run (see

Table 5: **Comparisons with simple Heuristic-based adaptation.** This table reports the average latency and energy factor for G2 in three co-run settings described in Figures 3, 6, and 7

Co-run Setting →		No other Apps		With Web Browser		With TikTok	
DNN Apps ↓	Metrics ↓	T&S (X=50)	DQN	T&S (X=50)	DQN	T&S (X=50)	DQN
YamNet	Time (ms)	3.690	4.250	127.120	20.482	53.400	23.050
	Energy Factor	0.461	0.502	15.890	3.392	4.429	2.707
SSD_MNv1	Time (ms)	8.260	6.170	191.740	8.248	60.670	30.860
	Energy Factor	1.161	0.734	24.351	1.284	8.037	3.890
WDSR	Time (ms)	126.290	6.750	56.320	5.042	142.000	114.220
	Energy Factor	15.701	0.714	8.599	0.729	18.034	3.948
Avg. Energy Factor		5.774	0.65	48.840	5.405	10.166	3.514

Figure 3), G2 co-run with a predictable App (see Figure 6), and G2 co-run with an unpredictable App (see Figure 7). Each execution of the DNN-based app conducts 250 inferences in total. Table 5 shows the results. The result shows that the simple adaptation by T&S is insufficient for fitting the continuously changing execution environments. The schedules it picks cause $3-10\times$ larger energy factors as well as frequently substantial slowdowns compared to the results by DQN.

7 Related Work

DNN workload under the stochastic runtime variance has been addressed in Autoscale [15]. The authors propose a lightweight scaling engine for DNN inference on a cloud-edge environment. It applies an offline-trained Q-table that observes DNN characteristics and runtime variance as states and selects execution targets as action. It is, however, only for single DNN execution.

Multi-tenancy DNNs [28] have been an active research topic in recent years. NestDNN [9] proposes an efficient scheduler that works with different model pruning ratios. The scheduling decision is guided by the proposed minimum total cost and minmax cost. The solution enhances the multi-DNN inference accuracy and video frame processing rate while reducing energy consumption. NeuOS [4] proposes a layer-by-layer multi-DNN scheduler. At each layer boundary, the system will determine the power configuration for each DNN based on their deadlines. Band [14] presents a model analyzer and scheduler to organize a multi-DNN workload on a heterogeneous platform. The model analyzer partitions multiple DNN models into several subgraphs and dynamically designates them with an eligible execution target. The scheduling decision is based on subgraph execution latency estimation using their tensor size and FLOPS.

Those solutions are however all centralized approaches, assuming the set of DNNs is fixed and there is a central runtime scheduler managing all the instances, making them inapplicable to the multi-instance DNN scheduling on open mobile systems.

8 Conclusion

This paper proposes the first-known decentralized application-level adaptive scheduler for multi-instance DNNs on open mobile devices. It builds on DQN, a reinforcement learning algorithm that actively explores and learns the relations between the states, actions, and rewards in a dynamic environment. The exploration uncovers a set of insights. It shows that it is possible for a decentralized scheduler to work effectively without direct knowledge of other apps in multi-instance DNN scheduling. The DQN-based scheduling works well regardless of the differences among underlying systems, hardware, and execution settings. The algorithm is shown to converge quickly and effective in improving co-run efficiency. As an application-level solution, it is ready to be immediately adopted across various mobile systems.

Acknowledgement

We thank Dr. Saurabh Bagchi for shepherding the preparation of the final version of this paper. This material is based upon work supported by the National Institute of Health (NIH) under Grant No. 1R01HD108473-01 and the National Science Foundation (NSF) under Grant No. CCF-2047516 (CAREER). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NIH or NSF.

References

[1] movenet/singlepose/thunder/4. Accessed on : insert access date.

[2] Tensorflow hub, 2022. Platform for hosting and serving machine learning models.

[3] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. https://tensorflow.org/, 2015.

[4] BATENI, S., AND LIU, C. Neuos: A latency-predictable multi-dimensional optimization framework for dnn-driven autonomous systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 371–385.

[5] BOCHKOVSKIY, A., WANG, C.-Y., AND LIAO, H.-Y. M. Yolov5. https://github.com/ultralytics/yolov5, 2021.

[6] BOWLING, M. Convergence problems of general-sum multiagent reinforcement learning. In *ICML* (2000), pp. 89–94.

[7] CHEN, X., AND DENG, X. Settling the complexity of two-player nash equilibrium. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)* (2006), IEEE Computer Society, pp. 261–272.

[8] DEVELOPERS, A. Android neural networks api. https://developer.android.com/ndk/guides/neuralnetworks/index.html. Accessed: January 10, 2023.

[9] FANG, B., ZENG, X., AND ZHANG, M. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (2018), pp. 115–127.

[10] FRIEDMAN, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[11] HAN, S., MAO, H., AND DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[12] HOWARD, A. G., ZHU, M., CHEN, B., KALENICHENKO, D., WANG, W., WEYAND, T., ANDREETTO, M., AND ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[13] HU, J., WELLMAN, M. P., ET AL. Multiagent reinforcement learning: theoretical framework and an algorithm. In *ICML* (1998), vol. 98, pp. 242–250.

[14] JEONG, J. S., LEE, J., KIM, D., JEON, C., JEONG, C., LEE, Y., AND CHUN, B.-G. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services* (2022), pp. 235–247.

[15] KIM, Y. G., AND WU, C.-J. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), IEEE, pp. 1082–1096.

[16] KOLOTOUROS, N., TZOUVARAS, P., WANG, X., ALAHARI, K., AND ZITNICK, C. L. Movenet: A video-based human motion estimation network. *arXiv preprint arXiv:2003.04831* (2020).

[17] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., ET AL. Human-level control through deep reinforcement learning. *nature 518*, 7540 (2015), 529–533.

[18] NIU, W., MA, X., LIN, S., WANG, S., QIAN, X., LIN, X., WANG, Y., AND REN, B. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 907–922.

[19] NIU, W., SUN, M., LI, Z., CHEN, J.-A., GUAN, J., SHEN, X., WANG, Y., LIU, S., LIN, X., AND REN, B. RT3D: Achieving real-time execution of 3d convolutional neural networks on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2021), vol. 35:10, pp. 9179–9187.

[20] PAGANI, S., MANOJ, P. S., JANTSCH, A., AND HENKEL, J. Machine learning for power, energy, and thermal management on multicore processors: A survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39*, 1 (2018), 101–116.

[21] SANDLER, M., HOWARD, A., ZHU, M., ZHMOGINOV, A., AND CHEN, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. *arXiv preprint arXiv:1801.04381* (2018).

[22] TAN, F. Some super resolution tflite models, 2022. GitHub repository.

[23] TAN, M., AND LE, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946* (2019).

[24] TEAM, G. A. Yamnet: An audio event recognition model. *arXiv preprint arXiv:1909.12916* (2019).

[25] VAN HASSELT, H., GUEZ, A., AND SILVER, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence* (2016), vol. 30.

[26] WANG, X., YU, K., WU, S., GU, J., LIU, Y., AND DONG, C. Esrgan: Enhanced super-resolution generative adversarial networks. *arXiv preprint arXiv:1809.00219* (2018).

[27] WATKINS, C. J., AND DAYAN, P. Q-learning. *Machine learning 8*, 3-4 (1992), 279–292.

[28] YU, F., WANG, D., SHANGGUAN, L., ZHANG, M., LIU, C., AND CHEN, X. A survey of multi-tenant deep learning inference on gpu. *arXiv preprint arXiv:2203.09040* (2022).

[29] YUN, S., SEO, S., KIM, J., AND LEE, K. M. Wide activation for efficient and accurate image super-resolution. In *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 557–572.

UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms

Ghazal Sadeghian Mohamed Elsakhawy Mohanna Shahrad[†] Joe Hattori[‡] Mohammad Shahrad

University of British Columbia *McGill University*[†] *University of Tokyo*[‡]

Abstract

We present UnFaaSener, a lightweight framework that enables serverless users to reduce their bills by harvesting non-serverless compute resources such as their VMs, on-premise servers, or personal computers. UnFaaSener is not a new serverless platform, nor does it require any support from today's production serverless platforms. It uses existing pub/sub services as the glue between the serverless application and offloading hosts. UnFaaSener's asynchronous scheduler takes into consideration the projected resource availability of the offloading hosts, various latency and cost components of serverless versus offloaded execution, the structure of the serverless application, and the developer's QoS expectations to find the most optimal offloading decisions. These decisions are then stored to be retrieved and propagated through the execution flow of the serverless application. The system supports partial offloading at the resolution of each function and utilizes several design choices to establish confidence and adaptiveness. We evaluate the effectiveness of UnFaaSener for serverless applications with various structures. UnFaaSener was able to deliver cost savings of up to 89.8% based on the invocation pattern and the structure of the application, when we limited the offloading cap to 90% in our experiments.

1 Introduction

Serverless computing [54] allows developers to quickly build scalable, event-driven applications and pay for only what they use. It also removes the provisioning and maintenance burdens of the traditional cloud system. Developers have identified and embraced this game-changing cloud paradigm. According to Datadog's June 2022 analysis of cloud user telemetry [33], more than 70% of organizations using AWS and over 50% of Azure and Google Cloud users have adopted serverless offerings. A year prior, the serverless adoption numbers for these three leading cloud providers were just above 50%, 35%, and 20%, respectively [32].

In addition to increased adoption, serverless applications are also becoming increasingly complex. In 2019, the majority of serverless applications were composed of just one function and 80% had three or fewer functions [72]. Today, complex serverless workflows are no more rare. A recent study of open-source serverless projects has identified 31% of studied applications to have workflow structures [35]. From 2019 to 2022 the popularity of serverless DAGs has grown by 6× at Azure [59]. The increased complexity of serverless applications can be attributed to the maturation of serverless offerings and the increased proficiency of developers.

With a developer-focused perspective, this work is motivated by a relatively simple question: *why should serverless functions be bound to be executed within the serverless platforms?* If serverless functions are designed to be primarily stateless, the serverless model disaggregates storage from compute, the serverless isolation/virtualization mechanisms are lightweight [17], and the model is event-driven, it begs asking whether offloading execution of serverless functions off the serverless platform can make economic sense.

Many organizations and development teams use serverless offerings in conjunction with other cloud service types, such as VMs or microservices' containers [24, 36]. Reports from different cloud providers indicate that the majority of VMs in public clouds are heavily under-utilized [30, 44]. Despite low VM utilization, public cloud providers have been able to improve data center efficiency using advanced resource oversubscription to co-locate many underutilized VMs with predictable guarantees [30, 48, 55], or through dynamic capacity harvesting from those underutilized VMs to sell to others [39, 74, 80]. Such strategies help the provider operate at higher efficiency, but cloud users still have to pay for their full static allocations. If a team is already paying an hourly rate for renting a VM and that VM is not fully utilized, it could be harnessed to run their own serverless functions. Additionally, an organization may have on-premise computational capacity that already incurs capital and operating costs, which can similarly be leveraged to execute migratable serverless functions.

The merits of the proposed serverless function offloading are clear, but determining when to migrate functions depends on various factors. For example, offloading one or more func-

tions in a latency-sensitive chain of short functions could lead to QoS violations due to added network latency. In contrast, for a serverless DAG with imbalanced branches, offloading those executions off the critical path may be worthwhile if the cost of data movement and added latency are acceptable. While these examples focus on latency, some serverless applications, such as nightly builds, may have little to no latency requirements, making offloading more viable. Ultimately, how and when functions of a serverless application can be offloaded depends on a long list of factors. Serverless providers do not have an incentive to enable such functionality as it would negatively impact their profitability, and developers would rather not deal with the complexity as it goes against the serverless philosophy of freeing them from provisioning concerns. We believe that there is real value to be delivered in this junction by providing developers with a system that adaptively and transparently offloads their serverless functions to their own alternative execution hosts.

We design and build UnFaaSener, **the first holistic serverless offloading system without any change to today's serverless platforms**. This system performs adaptive offloading of a developer's functions to their own alternative hosts. UnFaaSener does this by dynamically considering latency and cost implications of offloading as well as resource availability predictions on hosts against goals conveyed by the developer (e.g., saving maximum cost, or respecting a certain latency QoS). We build UnFaaSener to use existing services on a popular serverless platform, and run various serverless applications on it. It is available at https://github.com/ubc-cirrus-lab/unfaasener.

2 Background

2.1 The Status Quo

Execution of serverless functions. Your serverless functions run within the serverless platform you operate on. Depending on the provider, your functions might be allocated to run in lightweight VMs, containers, or other isolation abstractions that themselves use dedicated allocations or internally harvested resources [86]. Interestingly, your functions will not be allocated to any underutilized VMs that you already pay for. If you have computational resources on a different cloud or on-premise, those are not used to host your functions either. If a developer decides to tap into these capacities to reduce their serverless bill, they need to build their applications differently and effectively do resource provisioning. This defeats the purpose of using serverless in the first place.

Offloading to and from serverless. Serverless's unparalleled horizontal scaling and pay-per-use pricing model enables cheap acceleration of bursty, massively parallel workloads. Researchers have developed general purpose (e.g., gg [37]) and domain-specific frameworks (e.g., ExCamera [38] and NumPyWren [73]) for this purpose. Offloading to serverless is popular for edge [26, 52, 82] and network function virtualization (NFV) [16, 70, 85] applications. Researchers have also proposed offloading from serverless to the edge [41].

The idea of offloading from VMs to serverless has also been proposed. Most of these works utilize serverless as a backup when scaling out VMs [43, 47, 63, 84, 87], however, some others simultaneously offload a small portion of traffic to serverless [67]. As VMs are the primary deployment in these works, the benefits of serverless functions, such as high scalability can not be fully exploited, and the scope of the applications is limited to the capacity of the VMs. To fully exploit serverless advantages, researchers have suggested hybrid VM-serverless deployment of applications [56, 75]. In these systems, however, a secondary custom scheduler is added before the serverless scheduler, which limits the scalability of the system and comes with security concerns.

The systems mentioned earlier are not designed for resource harvesting. However, a number of works have proposed modifying the serverless platform to offload serverless on the harvested resources [78, 83, 86]. We identify this as a limitation, as one cannot expect serverless providers to change their platforms and reduce their profitability. Besides that, the offloading will be limited to hosts located only within the scope of the platform scheduler (same cluster, region, or zone of the same cloud, depending on the provider's architecture). UnFaaSener is designed to work with existing serverless platforms, without requiring any change. By harvesting the idle resources of any host within or outside the cloud hosting serverless functions, UnFaaSener opportunistically achieves cost reductions for a wide range of general-purpose applications, from single functions to applications in a form of complex DAGs (consisting of multiple branches, merging points, and dynamic fan-out patterns).

2.2 The Serverless Cost Model

Understanding the serverless cost model is imperative for building a mental model of how UnFaaSener offers cost savings. We provide a summary here and refer the reader to related work for more detailed descriptions [57].

Capacity cost: Developers are required to set the memory size of their serverless functions. This indirectly sets the CPU share, too, as the CPU-to-memory allocation ratio is fixed in current serverless systems [21]. Multiplying the execution time by the configured memory size determines the GB-seconds usage. There is a cost charged per GB-seconds; e.g., $\$1.67 \times 10^{-5}$ for AWS Lambda on x86 [3]. Some providers enforce a minimum execution time per invocation (e.g., 100 ms for Azure Functions). Execution times are also rounded-up; e.g., to the nearest 1 ms for Azure Functions and to the nearest 100 ms for Google Cloud Functions.

Invocation cost: Each invocation also incurs a cost; e.g., $0.2 and $0.4 per million requests for Azure Functions and Google Cloud Functions, respectively.

Free tier: Typically, serverless providers offer monthly free tiers: AWS, Azure, and Google Cloud provide 400,000 GB-

s per month. The free tier also includes no invocation cost for the first 1 million requests in AWS Lambda and Azure functions, and the first 2 million for Google Cloud Functions.

Saving cost by offloading: If a developer's serverless usage is low enough to fit within the monthly free tier offered by cloud providers, they should not run UnFaaSener. Beyond that, offloading functions to alternative hosts will eliminate the capacity and invocation costs associated with it.

3 UnFaaSener Design Challenges

Let us decompose the sub-problems that need to be solved to enable adaptive offloading of serverless functions to achieve maximum cost reduction with minimum impact on latency, and with no provider support:

1. ***Enabling flexible offloading*** (§4): The very first requirement is to enable partial offloading of requests for each function of a serverless application to arbitrary hosts. The solution should support various serverless applications, ranging from those with a single function to complex workflows (typically DAGs).
2. ***Asynchronous Scheduler*** (§5): To deliver a practical solution, UnFaaSener's design should satisfy the following:

(a) The system should work with no scheduling or load balancing support from the provider.

(b) As offloading is opportunistic, serverless execution should be the default case to ensure high scalability and low latency for the common case.

(c) The serverless platform should remain the end-point for the incoming traffic to the application to enforce network-based access control rules.

(d) Given that invocation patterns can be sporadic [72], continuously performing scheduling tasks on a dedicated VM/container is not an option, as the incurred cost can easily exceed the cost savings of function offloading.

We need a scheduling mechanism that resides outside the serverless platform, is activated when necessary, invokes the solver if needed for new decisions, and reflects the offloading decisions to be used by functions.

3. ***Determining optimal offloading*** (§6): Optimal offloading decisions depend on various factors such as application structure, function resource requirements, execution times, invocation patterns, host resource availability, communication latencies and costs. Additionally, the diverse range of developers using public clouds and the variety of serverless applications lead to different notions of optimality. Optimizing for latency vs. cost, mean vs. tail, etc., should all be expressible. A solver is required to take these inputs and determine the host(s) for function execution.
4. ***Monitoring and prediction of hosts' resource availabilities*** (§7.1, §7.2): Considering the delays associated with the distributed nature of UnFaaSener's multi-node, or even multi-cloud deployment, each host's resource availability should be forecasted with a high degree of confidence.

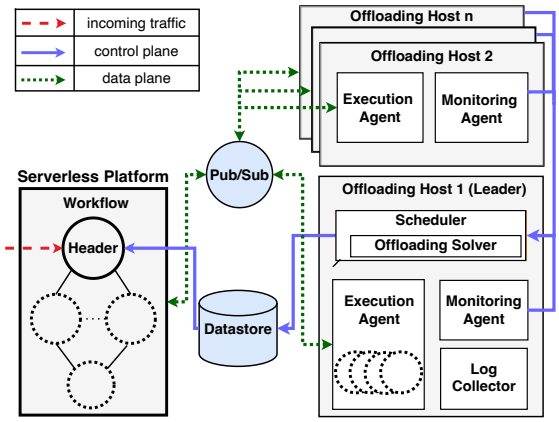


Figure 1: UnFaaSener's system diagram.

5. ***Managing execution-related tasks on the host side*** (§7.3): Each host is responsible for various mechanical tasks, such as pulling the function on a cold start, setting up an appropriate execution environment, enforcing resource limits, executing the function, implementing keep-alive policies, queuing incoming requests, and invoking the next function on the appropriate host. While this aspect of the system may not be particularly novel in terms of research, it is crucial for the overall performance of the system.

Figure 1 provides an overview of UnFaaSener's different system components. In the next sections, we describe how the design challenges stated earlier shaped our design decisions. In total, UnFaaSener includes ∼6.3 K-SLOC: 5.4K lines of Python, 0.8K lines of C++, and less than 100 lines of Shell.

4 Enabling Flexible Offloading

There is no universal approach to build and deploy a serverless application with more than one function. One can either use different messaging systems (e.g., AWS SNS), or rely on higher level abstractions delivered by services such as AWS Step Functions [5] and Google Workflows [10]. For us, it is critical to compose the application in a way that facilitates dynamically offloading arbitrary functions to user-specified hosts. Additionally, complexities of offloading functions to varying end-points, supporting dynamic DAG structures, or rate limiting should be kept hidden from the developers as much as possible. In this section, we explain how UnFaaSener achieves these goals. We motivate and explain design decisions that let us inject offloading decisions, propagate future decisions, and merge branches for DAGs. Throughout the section, we will use a simple example (shown in Figure 2) to demonstrate each implementation aspect.

Pub/Sub as the glue. Using a topic-based publisher/subscriber (pub/sub) messaging pattern enables us to control where each function has to be executed without changing

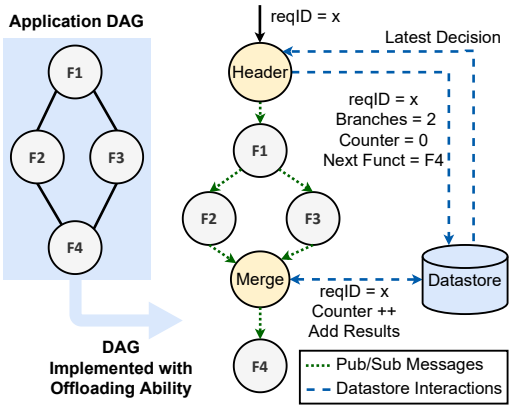


Figure 2: DAG conversion to support dynamic offloading.

the serverless platform. To do so, UnFaaSener first assigns a unique pub/sub topic to each offloading host (e.g., user's VMs) as a unique subscriber. Pub/sub's flexibility enables adding, removing, or hot-swapping hosts. This means an offloading host can be substituted by subscribing to the same topic without modifying pub/sub topics. Second, a few lines of code are inserted at the end of any non-terminal function to enable publishing to serverless or to any specific offloading host. We call this the *routing epilogue*. Finally, the routing decision for each function needs to be delivered to its parent function's routing epilogue. To do this, UnFaaSener adds a lightweight header function as the entry point of the DAG. Its role is to communicate the routing decisions made by the asynchronous scheduler to all functions by piggybacking the decisions to the incoming invocation.

The availability of pub/sub services across major clouds (e.g., AWS SNS and Google Pub/Sub) and strong support for it in various programming languages was a major driving factor for us to facilitate portability of UnFaaSener. Furthermore, the use of pub/sub provides a degree of fault tolerance by requiring subscriber acknowledgment. If no acknowledgment is received within a set timeframe, pub/sub automatically tries to resend the message. We evaluate pub/sub latency in §9.8.

Routing epilogue. The routing epilogue is a general code snippet added to the end of all non-terminal functions in the application. The code snippet in Figure 3 shows the routing epilogue for a Python function. The routing epilogue adds only a conditional statement in the critical path of execution, resulting in negligible added latency compared to the regular call to the subsequent function. Based on the routing character sent to this function by the header function, the role of this code piece is to route the invocation to a serverless endpoint or to any offloading host. The former is encoded by a `"0"` character, and the latter is determined by the host ID embedded in the character. This works as UnFaaSener names pub/sub topics for offloading hosts to follow the same convention: e.g., `hostTopic1`, `hostTopic2`, etc.

Header function. As illustrated in Figure 2, UnFaaSener adds a header function to the head of the DAG. The role of the header is to generate the routing decisions to be used by the

```
...
if (routing == "0"):    # run next function in
     serverless
    topic = publisher.topic_path(projectID, "
    F3")
    publish_future = publisher.publish(topic,
    data=message, reqID=reqID, routing=
    routingData.encode('utf-8'))
    publish_future.result()
else:   # offload next function to a host
    hostNumber = ord(routing) - 64
    hostTopic = "hostTopic" + str(hostNumber)
    topic = publisher.topic_path(projectID,
    hostTopic)
    publish_future = publisher.publish(topic,
    data=message, reqID=reqID, invokedFunction
    ="F3",routing=routingData.encode('utf-8'))
    publish_future.result()
```

Figure 3: The routing epilogue appended to a function.

routing epilogue of non-terminal functions. It also takes care of creating entities for the merging points, as described next.

Merge function. In a generic serverless DAG, a function may have several predecessors. A challenge is dealing with predecessors finishing at different times. Since serverless functions are primarily stateless, we need to persist information sent by predecessors. One way to solve this is using stateful functions, such as AWS Step Functions [5] and Azure Durable Functions [22], to join on predecessors. To provide a merging mechanism that works for predecessors executed on different hosts UnFaaSener uses a database; specifically, a NoSQL database (Google Datastore) in our current implementation.

As shown in Figure 2, the header function creates a new entity (data object) for each merging point in the Datastore by passing the metadata containing the request ID, the number of branches, and the subsequent function to be invoked after that merging point. The merge function, which is added between the predecessors and the child function, is invoked by each predecessor. It keeps track of the content and number of responses from the predecessors using the Datastore entity. It determines the completion of the last predecessor by comparing the number of times it was invoked by the same request ID to the predecessor count stored for it by the header function in Datastore. The merge function then triggers the next function and removes that entity from the Datastore.

DAG description. The developer provides UnFaaSener with a DAG description JSON file, which defines the structure of the DAG. The code snippet in §A.2 shows the workflow description for one of our benchmark applications.

5 Asynchronous Scheduler

In this section, we describe UnFaaSener's scheduler and how we address the challenges stated earlier in §3.

An asynchronous scheduler off the critical path. To address challenges 2(a), 2(b), and 2(c), we design UnFaaSener's scheduler to only activate when making offloading decisions.

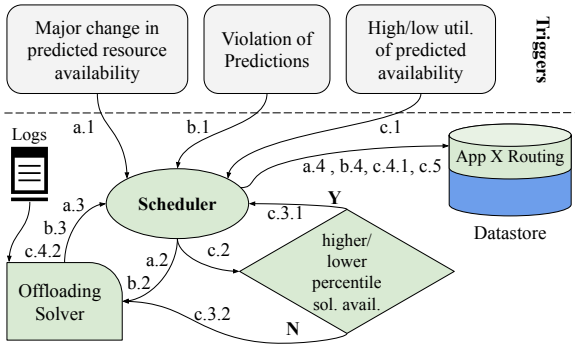


Figure 4: Different events triggering the asynchronous scheduler alongside the decision-making flow for each.

New offloading decisions do not need to be determined for every single invocation, and the header function (§4) can rely on the most recent decisions available in the Datastore.

Run in an offloading host. To run UnFaaSener, one or more offloading hosts are needed. Otherwise, there is no offloading and subsequently no cost reduction. Now, if there are already offloading host(s) available, we can leverage them to host the asynchronous scheduler as well. This addresses challenge 2(d) stated in §3. When there are more than one offloading hosts, the developer can tag one of them as the leader at the setup stage. If the leader host is not specified by the developer or if the leader fails, we use the Raft [64] consensus algorithm to elect the next leader in the order of the highest average available resources since deployment.

Triggering rules. The offloading solver, which will be presented in the next section, determines the most optimal offloading decisions theoretically. However, serverless workloads can be highly dynamic. The request inter-arrival times are shown to be highly variable [72], function execution times for certain applications can be heavily input-dependent [62], and the relatively short median execution time for serverless functions [32, 72] amplifies the relative performance jitter in the presence of third-party API calls. The resource utilization of offloading hosts can change frequently, as well. To prevent the solver from being constantly re-invoked, we pick minimal triggering rules and rely on pre-solved scenarios. Figure 4 shows how system events trigger the scheduler.

A common case requiring new offloading decisions is variations in the invocation rate. UnFaaSener's Log Collector (§8) maintains an updated view of each application's invocation rate distribution on the leader host. Whenever the offloading solver is run, it solves the same problem for different percentiles of the invocation rate distribution: 25^{th}, 50^{th}, 75^{th}, and 95^{th} percentiles. The scheduler uses the 50^{th} percentile solutions for determining offloading decisions. Later, if the utilization of predicted available resources was too high (85%) or too low (20%), the scheduler can quickly pick the solution for a higher or lower invocation percentile, respectively. We found this simple mechanism effective for rapid flow control without re-running the solver. In the high load case, if the offloading decisions assuming the 95^{th} percentile rate is still not low enough, the solver is triggered.

6 Offloading Solver

In §3, we discussed the offloading solver that sits at the core of UnFaaSener and is responsible for determining which functions to offload, to where, and to what degree. At a high level, the solver is a non-linear optimizer that considers execution times, latencies, costs, resource demands, and availabilities to determine the optimal offloading decisions. We describe various design choices for UnFaaSener's solver in this section.

Supporting different optimization goals: Any optimization has an objective function. As one can imagine, the wide range of service-level objectives (SLOs) in public cloud offerings and the broad spectrum of serverless applications (e.g., from latency-sensitive speech recognition [46] to cost-oriented nightly builds [58, 60]) prevents using a one-size-fits-all objective function. As a result, we built UnFaaSener's solver to support two different modes of operation: the **cost mode** and the **latency mode**. In the cost mode, the goal is to get maximum cost reduction without considering any constraints on the added latency. In the latency mode, the solver aims to optimize for maximum cost reduction in the face of a specific tolerance window for the added latency. By default, the tolerance window is set to the median latency of the workflow (the start of the first function to the end of the last function) when executed fully on serverless. This value is easily modifiable by the user, and we evaluate the impact of changing it in §9.3. Users have the flexibility to modify existing optimization modes or introduce their own.

Considering locality: The solver can be re-invoked based on changes in the resource availability of the hosts, change in traffic patterns, etc. Each invocation of the solver should not result in vastly different offloading decisions. Instead, the solver should consider the current offloading host(s) of each function and minimize migrations as much as possible. This is because offloading a function to a new host would incur additional latency to pull the code and build a container image. The solver has a locality parameter, α, to control the degree of emphasis placed on the locality.

Robustness to performance variations: The solver consumes the data gathered by the Log Collector (§8) to determine the most optimal offloading. These logs contain observations for execution times on the serverless platform and various hosts, as well as communication latencies. Execution time for serverless functions has been shown to be highly variable [42, 66]. The latency of pub/sub, which we use to build our flexible offloading framework, can be highly variable too (we evaluate this in §9.8). In this context, using only the mean or median of limited observations can potentially misguide the solver, specially in the latency mode, where a latency QoS should be respected. Thus, the solver first checks the similarity between serverless and host execution time distributions using the Kolmogorov–Smirnov (KS) test, a non-parametric

method that makes no assumptions about the normality of the data distributions. If the KS test reveals distinct distributions for more than 70% of offloaded workflow functions, the solver uses mean statistics. However, if the distributions are not distinguishable for more than 30% of offloaded functions, the solver solves for the following three scenarios using non-parametric confidence intervals (95% confidence level [61]) and reports the average of three solutions as the final answer:

1. **Best case:** This is to model the situation where everything goes well for offloading to hosts. The lower confidence interval bound for each host's execution records, and upper confidence interval bound for serverless execution records are used by the solver.
2. **Worst case:** Here, confidence intervals are chosen so that we account for the low-end cost reduction and high-end added latency for offloading each function. Thus, the choices for confidence interval bounds are opposite to the best case mode.
3. **Mean case:** Here, the mean of system logs for each unique function-host pair is used.

The total offloading decision percentage for each function is limited to 90%, whether using a single or triple solver. Keeping some traffic on the serverless platform allows for 1) continuously observing execution times and latencies in spite of a varying workload, and 2) keeping some serverless function images warm in case the host(s) becomes unavailable.

Multi-host offloading: When multiple hosts are available, maximum cost reduction may require offloading the same function to more than one host. In such a case, each host will be in charge of a portion of the incoming traffic to that function. Here, the solver considers the resource availability of multiple hosts and recommends partial (as opposed to binary) offloading decisions.

Optimization formulation: We have reviewed various design aspects of the solver. Let us go over the formulation of the optimization problem, shown in Algorithm 1. The latency mode optimization has an additional constraint (Constraint 4) for comparing the added latency with the latency tolerance.

Intuitively, the solver aims to find offloading decisions that lead to maximum cost reduction, without violating resource or QoS constraints. Offloading decisions are partial (as opposed to binary) to ensure delivering cost savings even with limited host capacity. Cost is defined as the serverless execution cost, which the solver can try to minimize; unlike the host cost, which is already paid for. The predicted serverless cost is based on the invocation rate and execution time history, as well as the capacity required by the function. Therefore, the solver will prioritize offloading the function with higher resource usage, higher execution time, or higher invocation rate, as it will result in a higher cost reduction.

Implementation: We implemented the solver in Python to leverage its rich optimization and data manipulation packages. We used the GEKKO [19] package for mixed integer nonlinear programming (MINLP), used the CriticalPath [76] package for slack analysis, and used the Pandas [1] package for storing logs and efficiently performing complex statistical operations.

Algorithm 1 Optimization algorithm used by the Solver.

α: Locality Weight
μ: Adjusted Average CPU Utilization
$d^t_{n,i}$: Func_i offloading percent on host_n at time_t
rps_i: Request per second for Func_i
$\text{ExecTime}_{n,i}$: Execution time of Func_i on host_n
Scenarios: All offloading scenarios based on partial decisions.
host_n **or** serverless $\leftarrow$ scenario$[i]$ ▷ The assigned placement for Func_i in a scenario.
Directed Paths: All paths starting from an initial node and ending at a terminal node in a DAG
$\text{CommLatency}_{\text{scenario}[m],\text{scenario}[n]}$: Communication latency between Func_m on scenario$[m]$ and Func_n on scenario$[n]$.
$\text{Slack}_{\text{path}} = \text{Duration}_{\text{critical path}} - \text{Duration}_{\text{path}}$
$\text{ExecLatency}_{n,i} = \text{ExecTime}_{n,i} - \text{ServerlessExec}_i$ ▷ Added latency by executing Func_i on host_n versus serverless execution.
procedure CALCCOST($d^t_{n,i}$)
  Cost = 0 ▷ Assumes hosts with fixed cost regardless of utilization, e.g., a VM billed at an hourly rate.
  for $\text{Func}_i \in$ functions **do**
    $\text{offloading}_i = 0$
    for $n \in$ offloadingHosts **do**
      $\text{offloading}_i \leftarrow \text{offloading}_i + \frac{d^t_{n,i}}{100}$
      $\text{Cost} \leftarrow \text{Cost} + \alpha \times \left|\min(d^t_{n,i}, 1) - \min(d_{n,i}{}^{t-1}, 1)\right|$
    $\text{Cost} \leftarrow \text{Cost} + (1-\alpha) \times \text{cost}_{\text{Func}_i} \times \text{rps}_i \times (0.9 - \text{offloading}_i)$
Constraint 1: $\sum_{\text{Func}_i \in functions} \mu \times \text{ExecTime}_{n,i} \times \text{rps}_i \times \frac{d^t_{n,i}}{100} \leq \text{AvailableCPU}_{\text{host}_n}$
Constraint 2: $\sum_{\text{Func}_i \in functions} \text{Mem}_i \times \text{ExecTime}_{n,i} \times \text{rps}_i \times \frac{d^t_{n,i}}{100} \leq \text{AvailableMem}_{\text{host}_n}$
Constraint 3: $\sum_{n \in \text{offloading hosts}} d^t_{n,i} \leq 90$
Constraint 4: (only for the latency mode)
for scenario $\in$ Scenarios **do**
  for path $\in$ Directed Paths **do**
    latency = 0
    for $\text{Func}_i \in$ path **do**
      if scenario$[i]$!= serverless **then**
        $\text{host}_n \leftarrow$ scenario$[i]$
        $\text{latency} \leftarrow \text{latency} + \min(d^t_{n,i}, 1) \times \text{ExecLatency}_{n,i}$
      for $\text{Func}_j \in$path **and** $\text{Func}_j \in$predecessors$_i$ **do**
        $\text{latency} \leftarrow \text{latency} + \text{CommLatency}_{\text{scenario}[i],\text{scenario}[j]}$
    $\text{latency} \leq \text{Slack}_{path} + \text{LatencyTolerance}$
Decision: $d^{*t}_{n,i}$ (OptimalValue) $\leftarrow \underset{d^t_{n,i}}{\text{argmin}}$ CALCCOST($d^t_{n,i}$)

7 Host Agents

Let us explain UnFaaSener's different host agents.

7.1 Resource Monitor Agent

Implemented in C++, this lightweight agent tracks the CPU and memory usage at the host. It distinguishes between UnFaaSener-related processes (including containers running offloaded functions) and host processes to create differentiated scheduler triggers, described in §5. The monitoring period is 100 ms in our implementation

7.2 Resource Predictor Agent

The predictor agent, also written in C++, is tightly coupled with the monitor agent. It periodically (every 1 s in our implementation) predicts the maximum resource utilization in the next time window. UnFaaSener's effectiveness in cost reduction is as good as the accuracy of the predictions made by this agent. If predictions are too conservative, the host's available capacity will not be well utilized, limiting cost reductions. Conversely, aggressive predictions risk causing resource contention between offloaded functions and host processes. We formulate this trade-off using the following metrics:

1. Reclamation Efficiency (RE): *RE* represents how much of the available resources could be reclaimed based on the predicted usage. If predictions always match the peak resource usage, *RE* would be 100%.
2. Violation Rate (V): If the predicted usage in a window is lower than the materialized peak usage, the *RE* is capped at 100% and a prediction violation event is logged. *V* denotes the percentage of predictions leading to a violation.

An ideal predictor would yield $RE = 100\%$ and $V = 0\%$. For each resource dimension (e.g., CPU and memory), the objective function combining the two looks like this:

$$PredictionScore = w \times RE + (1-w) \times (100 - V) \quad (1)$$

Here, w denotes the resource reclamation weight. By default, UnFaaSener gives equal importance to reclamation efficiency maximization and violation minimization ($w = 0.5$) as it does not make any assumptions about the host workloads. Further knowledge about the workload or user's tolerance of slowdown for it can change this. We do not explore this angle in this work and only use $w = 0.5$. We evaluate the prediction quality and performance of various prediction policies in §9.9 and derive the one best suited to UnFaaSener.

7.3 Execution Agent

The execution agent subscribes to the pub/sub topic of its host and is notified on incoming invocations. If that function's code is not present on the host, the execution agent proceeds to download the function's code and metadata (runtime and memory limit) with a call to the Google Cloud Functions' API. Once the function's code and metadata are retrieved, a new docker container image is built using the skeleton container for the specified runtime (e.g., Python 3.10). We use Docker Hub [8] to host skeleton images. The agent leverages Docker's build utility to generate a local image of the runtime that contains the function's code and dependencies. As the image generation step is computationally expensive, the generated images are kept on the host for future use. Prior work has shown that even employing simple keep-alive policies can notably reduce cold starts [40, 69, 72]. The execution agent stops an idle container when the time since the end of the last execution exceeds the keep-alive window (10 minutes), or when capacity is needed to execute a different function.

UnFaaSener's execution agent supports concurrent execution of multiple instances of the same or different functions. However, the agent queues incoming requests on the host if the predicted resource availability is more than 90% utilized, or if the current degree of concurrency is at or beyond the concurrency limit. The execution agent has a feedback mechanism to set the concurrency limit dynamically based on observed CPU utilization and performance degradation of offloaded functions. It starts with assuming one CPU thread per function, the concurrency limit is thus set to the number of cores available in the host. Over time, the agent has access to average container CPU utilization reported by the monitoring agent. It also has access to execution time trends for functions allowing it to measure any slowdown. Combining the two, it calculates the adjusted average CPU utilization as:

$$\mu = \min(1, \text{AvgFuncCPUUtil} + 0.03 \times (e^{\Delta \text{ExecTime}} - 1)) \quad (2)$$

From this, the concurrency limit is calculated as $\frac{\text{Core Count}}{\mu}$. The insight for this asymmetric feedback mechanism is as follows: if no performance degradation is sensed, μ directly reflects the average utilization of docker containers hosting offloaded functions. However, with slight degradation, μ is increased super linearly to reduce the concurrency limit. More details on this process is provided in §A.1.

UnFaaSener's approach of using pre-solved offloading decisions accelerates the scheduling path. This approach also makes the scheduler early binding [50]. The distributed nature of the system introduces some delay from when a burst hits to the time that the host scheduler updates the offloading decisions. This can create a request build-up on the hosts' queues. To prevent QoS violations, the execution agent monitors requests in the queue and if their wait time exceeds a specified window (2 s by default), that request and all its descendant function calls are redirected to the serverless platform.

In §4 we described how using the subscription retry policy of pub/sub can provide a degree of fault tolerance. If no acknowledgment is received within the specified number of retries (default 5), the message is redirected to the subscriber's *dead-letter* topic, provided by pub/sub. We have designated the topic of $host_{(i+1)\%num_{hosts}}$ as the *dead-letter* topic for $host_i$. Additionally, to prevent duplicate execution on the host side, we have made use of the *exactly-once delivery* feature. However, our current implementation lacks complete handling of a host failure scenario. Although no further offloading occurs to the failed host, any invocations previously acknowledged

and queued on that host will be lost.

Offloaded function execution logs are stored locally and periodically published to a Datastore entity every one thousand logs for backup. This enables a smooth transition to a new leader host in case of primary host failure.

8 Log Collection

The log collector runs periodically (every 1 minute in our implementation) on the leader host, collecting execution and latency logs for serverless as well as host executions logs. The logs for serverless executions are collected using the Google Cloud CLI (gcloud [27]). Aside from execution time information, serverless logs contain timestamps for the invocation of the header function that runs exclusively on serverless. These timestamps are used to calculate the invocation rate distribution for the workflow, which is used by the solver.

Host execution logs are needed by the solver to enforce the latency tolerance QoS (discussed in §6). As mentioned in §7.3, the execution agent writes execution records (start time, end time, invocation ID, and function ID) to local log caches, and asynchronously backs them up on the execution log Datastore entity. In the event of leader failure, the log collector retrieves execution records from the Datastore entity and stores them in a pandas dataframe, similar to how local log caches store host logs. This stored data is readily accessible for the offloading solver upon activation. To accelerate statistical operations, the dataframe stores a maximum of N most recent data points per function-host, with $N = 50$ yielding favorable results in our experiments.

9 Evaluation

9.1 Setup, Methodology, and Benchmarks

UnFaaSener has many components and serverless settings are complex. We try to carefully distill various design and performance aspects of UnFaaSener without confusing ourselves or the reader with unnecessary complexity. Each experiment conveys specific points to help readers build a mental model. The majority of our evaluations are from real deployment, but we also conduct some simulations to stress certain parts of the system with more usage scenarios. Any reported cost normalization is based on the scenario where all the functions of the workflow run on serverless.

9.1.1 Benchmark Applications

We used five real-world serverless applications to evaluate UnFaaSener; 1) DNA Visualization [28], a script that performs visualization of the input DNA sequence file, 2) Image Processing [51], an application that performs a sequence of operations on input images, 3) Text2Speech Censoring [34] turns short text segments into speech and censors any profanities within the text segment, 4) Regression Tuning [12], which solves a regression problem using Keras [25], and 5) Video Analytics [13], an application that performs object recognition on images generated from a video stream. Table 1 shows the diverse range of structures covered by these applications. The dynamic fanout column in the table captures whether the application has a sub-graph with parametrized fanout.

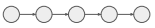
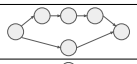
Benchmark	Branch	Dyn. Fanout	DAG Structure
DNA Visualization	✗	✗	
Image Processing	✗	✗	
Text2Speech	✓	✗	
Regression Tuning	✓	✗	
Video Analytics	✗	✓	

Table 1: Benchmark applications were chosen to represent various structures present in today's serverless.

9.1.2 Workload Invocation and Traffic

We use FaaSProfiler [9, 71], a serverless testing tool used by prior work [50, 66, 72], to invoke traffic patterns precisely. Depending on the nature of the experiment, we use different invocation patterns. For evaluating high-level trade-offs or assessing the extremes, using a uniform invocation rate suffices. For those with co-location scenarios, we use the 2021 Azure Functions Invocation Trace [6, 86].

9.1.3 Ensuring fair comparisons

We take the following steps to make sure that our reported gains are not inflated: 1) All cost numbers include the cost of functions added by our system; 2) all latency numbers are end-to-end and include the latency overhead introduced by UnFaaSener; 3) each function is tuned [2] for the most cost-optimal memory configuration. By making the baseline serverless functions as cost-efficient as it gets, we ensure that UnFaaSener's cost savings are not an artifact of comparing to bloated functions; 4) when comparing UnFaaSener to alternative solutions (§9.5), we ensure that each implementation is minimal and tuned to the specific offering; 5) we invoke each application with a set of random inputs; 6) we ignore the bootstrapping phase measurements, as UnFaaSener tends to offload more during this phase, leading to more cost savings.

9.2 Latency Mode vs. Cost Mode

Here, we use a VM with 4 vCPUs and 16 GB of memory to demonstrate how even a relatively small host can be used by UnFaaSener to offer cost savings. Figure 5 shows the normalized execution cost and end-to-end latency values associated with running three applications in latency and cost modes. For each application, we chose the invocation frequencies such that the maximum offloading (with a low rate) and minimum offloading (with a high enough rate) are stressed.

As expected, using the cost mode leads to more savings than the latency mode. It comes at the expense of potentially increased latency since the solver considers no latency constraint when making offloading decisions in the cost mode.

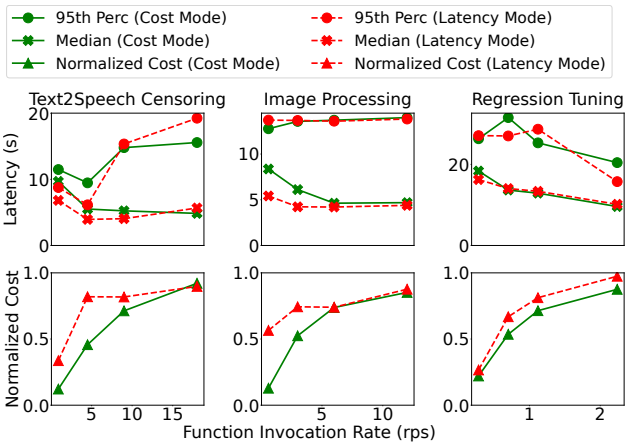


Figure 5: UnFaaSener's cost savings and end-to-end latency in two optimization modes.

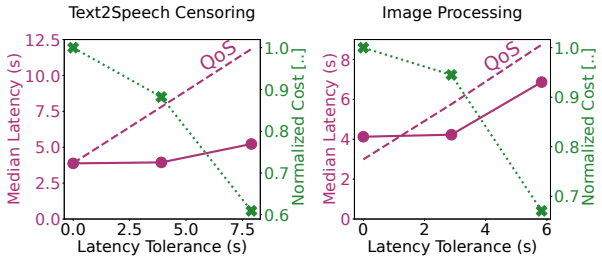


Figure 6: Increasing the latency tolerance in the latency mode leads to more offloading, and thus more savings.

Even in the latency mode, UnFaaSener can reduce the cost, but is limited by the default latency tolerance. As the invocation rates increase, the cost savings achieved by UnFaaSener decrease. This is because UnFaaSener is designed to harvest the unused computing resources of already-allocated hosts. Therefore, when the invocation rates are high, the system is fully utilizing the resources, leaving little or no unused capacity to be harvested.

Later in §9.3, we show how changing the latency tolerance affects the results in the latency mode.

We measured the added latency of the header function to be ∼50ms. The end-to-end latency values include this overhead.

9.3 Latency Mode and Tolerance Window

In the latency mode, the solver tries respecting a tolerance on the added latency when making offloading decisions. Figure 6 shows the effect of changing the tolerance window on the cost (green cross markers) and median latency (purple circle markers). We studied its effect on two of our benchmarks: Image Processing and Text2Speech Censoring. The tolerance windows are set to zero, the median, and twice the median latency of each workflow when solely run on the serverless platform (baseline latency). The QoS (baseline + tolerance) for these applications is depicted with purple dashed lines.

Overall, UnFaaSener manages to offload in a controlled fashion and complies with the set tolerance window; except for when latency tolerance of 0 is set for Image Processing. Image Processing is a chain and thus every offloading is on the critical path (see Table 1). This is unlike Text2Speech that has asymmetric branches, because of which it enjoys higher cost reductions and a higher safety margin from the QoS line.

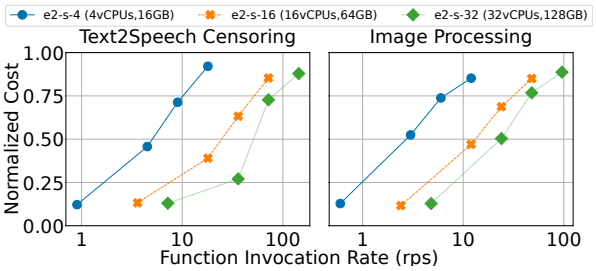


Figure 7: UnFaaSener leverages larger hosts effectively to increase offloading (reduce cost).

9.4 Impact of Host Size

The results presented so far were gathered on a relatively small host. This was to show that even with a small host, or equivalently, with a small leftover capacity, UnFaaSener can offer cost savings. It is worth asking how those results scale if we use a larger host. To answer this, we scale up the cost-mode experiments presented in §9.2 on two larger hosts. To prevent factors other than the resource capacity, we picked larger hosts from the same VM family as the small host (Google Cloud's *e2-standard* family). For brevity, we only show the cost saving results for two benchmark applications in Figure 7. The X-axis is logarithmic, and multiplying invocation rates appears as a shift to the right. As seen, with increased host size, the cost saving curve is shifted to the right consistently.

9.5 Comparison to Alternative Solutions

The primary objective of UnFaaSener is reducing the cost by offloading to pre-paid hosts. To evaluate whether we accomplished this goal, we compare the maximum cost savings, which is also the worst case latency, offered in the cost mode with two popular serverless workflow platforms: AWS Step Functions and Google Workflows. We also compare it to using AWS Lambda functions glued with AWS SNS, and Google Cloud functions glued with Google Pub/Sub. The latter is the underlying setup for UnFaaSener. Furthermore, we present results for UnFaaSener's latency mode using the default tolerance window. Figure 8 compares the average latency and cost per invocation for each benchmark and platform. We could not port Text2Speech Censoring benchmark to AWS as this benchmark utilizes Google Translate's text-to-speech API [11]. The cost mode is all about cost reduction. We see that UnFaaSener is able to trade latency with cost effectively, lowering the cost by about an order of magnitude. Using the latency mode limits this cost reduction depending on the latency tolerance expressed by developers (§9.2 and §9.3), and helps cut down on average latency compared to the cost mode, but the cost savings are not as significant. It is worth noting that DNA Visualization, a single-function application, experiences no added latency as it has low coordination overhead.

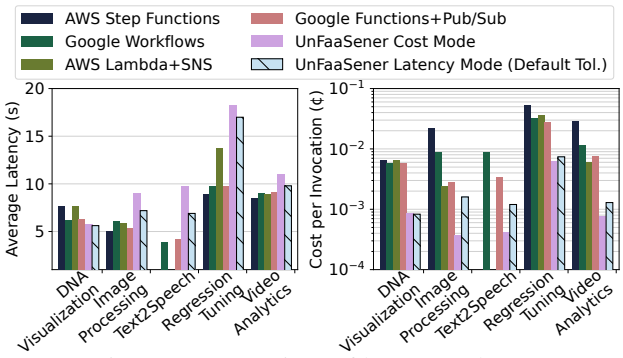


Figure 8: Comparison of latency and cost.

Finally, we see that using workflow coordination services comes with major cost implications. This is because, in addition to the capacity and invocation cost elements mentioned in §2.2, the developer has to pay for state transitions in these offerings [4, 15].

9.6 Adaptive Cost Saving

We are interested in assessing the responsiveness of UnFaaSener to host processes. We use the host with 16 vCPUs from earlier in this experiment. We replay a sample trace from the 2021 Azure Functions traces [6] using FaaSProfiler [9] to invoke the Image Processing application. At second 110, the Graph Analytics workload from CloudSuite 3.0 [7, 65] is run on the host for about 80 seconds. The workload uses Apache Spark to perform graph analytics on a large-scale Twitter dataset and uses all 16 vCPUs as well as 15 GB of memory. Before and after this window, the host is mostly idle and fully available for offloading.

Figure 9 shows the cost for each execution of the Image Processing workflow over time. The timespan for the execution of the heavy host workload is marked with dashed lines. Soon after 110 s, the execution agent slows down admitting new requests and the monitoring agent triggers the scheduler with a prediction failure trigger due to a sudden host load spike. New predictions are made, and the solver makes new offloading decisions for minimal offloading. This is reflected in increased cost during that period. After the VM workload ends, the predictor remains cautious briefly before declaring the majority of the host's resources as available. Offloading to the VM resumes as before. The two orders of magnitude difference in cost is attributed to the non-offloadable header function taking less than 100 ms to execute, while the rest of the application takes 3-4 seconds.

9.7 Host Interference

To measure the impact of UnFaaSener on host processes, we run Text2Speech and Image Processing applications on the 16-core host described in §9.4. We invoke them with the similar rates to those used in that section, and adhere to using the cost mode which guarantees maximal offloading, and equivalently, maximum host-side interference. We use three standard benchmark applications on the host to measure the impact

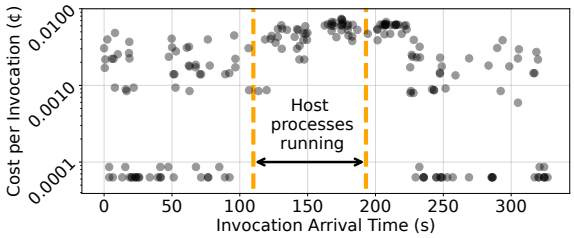


Figure 9: UnFaaSener adaptively adjusts offloading to respect host processes.

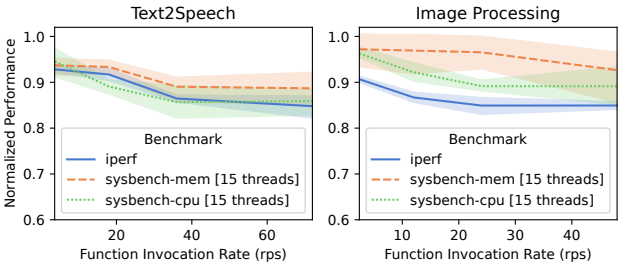


Figure 10: Degradation of host processes is a function of traffic rate and the resource requirements of the functions.

on CPU, memory bandwidth, and network performance. For the first two, we use Sysbench [53], in CPU and memory test modes, respectively, each with 15 active threads. For network, we use the Iperf [79] benchmark. Figure 10 shows the performance of these benchmarks when normalized to that of an idle host without UnFaaSener running. Across experiments, the maximum average degradation at maximum offloading is less than 15%. We find this degradation reasonable considering that 1) benchmarks used are highly sensitive and 2) the developer is in the loop and aware of the operation of UnFaaSener resource harvesting. We observe that sysbench-mem benchmark experiences significantly higher degradation for the Text2Speech serverless application. We pinpointed this to Text2Speech having a function with a 2 GB memory configuration, whereas the largest function for Image Processing requires 256 MB of memory.

9.8 Pub/Sub Latency Overhead

Using pub/sub allows us to invoke functions on and from offloading hosts. Here, we characterize the latency overhead of the Google Pub/Sub [14], used by UnFaaSener.

In our experiment, the publisher is a Google Cloud function in the East Coast sending payloads to three subscribers: 1) another Google Cloud function in the same region, 2) a Google Cloud VM in the same region, and 3) a private VM in the West Coast. We test each publisher-subscriber pair with two message sizes (10 KB and 1 MB) and two invocation rates (0.1 rps and 5 rps). These values were driven by a recent characterization of production serverless DAGs [59]. We chose 10 KB and 1 MB message sizes to estimate high-end values for regular and high-fanout DAGs [59], respectively. Similarly, 5 rps and 0.1 rps rates were chosen as high and medium average invocation rates based on that characterization. We collected 250 samples per scenario (3,000 samples in total).

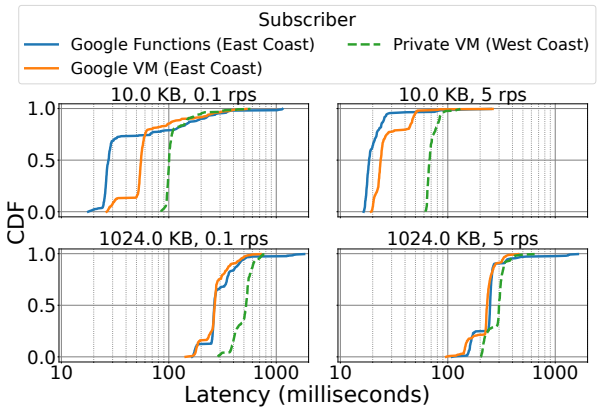


Figure 11: Pub/sub latency depends on message size, subscriber type, distance, and messaging rate.

Figure 11 compares the distribution of latency depending on the subscriber. Higher message size (1 MB vs. 10 KB), longer distance (West vs. East Coast), and very low invocation rate (0.1 rps vs 5 rps) increase the latency. The latter is likely caused by higher chances of lost connections between the publisher, forwarder, and subscriber during longer invocation periods. The wide distributions in Figure 11 reveal Pub/Sub's high performance variability. Even for two Google Cloud functions in the same region, the latency varies from less than 20 ms to more than a second. Such non-deterministic variations in Pub/Sub latency alongside high performance variations of serverless functions motivated us to use confidence intervals statistics for the solver.

9.9 VM Resource Prediction Policy

The predictor agent runs on each host to forecast the maximum resource utilization anticipated for host processes. It has two predictors: one for CPU and another for memory.

We dedicate this section to compare various prediction policies and determine their fitness to be used by the Predictor agent. To assess prediction policies for a wide range of VM usage scenarios we rely on simulations. We use the Azure Public Dataset [29, 30] to simulate different prediction policies for 1 million VMs over a 30-day period. The dataset includes 5-minute VM CPU utilization readings (min, average, and max utilization per reading) and has no memory readings. CPU utilization is inherently more variable than memory utilization. Besides, CPU utilization percentages can experience a wider variation range due to CPU being the typical resource bottleneck in public clouds [45]. These factors make forecasting CPU utilization a harder task.

We implement eight common time series forecasting methods and use traces from 100,000 VMs to train the best policy parameters for them. These trained policy parameters are used as the initial parameters for test VMs and during the lifetime of each test VM the parameters are periodically retuned per VM. A brief description of explored methods and their corresponding parameters (shown in curly brackets) is listed below. For all policies, we also consider a safeguard margin (m) to allow exploring conservative predictions. To derive this margin despite inherent differences between various prediction methods, we also included m as a training parameter.

Simple Exponential Smoothing (SES) {pars: m, α}:

$$s_{n+1} = \alpha x_n + (1-\alpha)s_n,\ x_{n+1} = s_{n+1} \times (1+m) \quad (3)$$

Simple Moving Average (SMA) {pars: m, N}:

$$x_{n+1} = \frac{1+m}{N} \sum_{n-N+1}^{n} x_i \quad (4)$$

The averaging window is limited to the existing number of observation if there are less than N observations.

Histogram (Hist) {pars: m, p, d}: Recent work [68, 72, 81] has demonstrated the superior performance of histogram-based time series forecasting. We implemented a histogram with a 1% utilization resolution. At each prediction window, the max observed utilization observed gets rounded to determine the histogram bin to be incremented. A certain percentile of the histogram (p) is then used to determine the prediction for the next window. A high percentile reduces violations, but reduces reclamation efficiency. Old observations in the histogram are depreciated using the decay factor (d).

Markov Chain (MC) {pars: m, r, o}: Markov Chains have been used for time series forecasting in various problem domains [18, 23, 49]. We build a simple MC predictor for CPU prediction. A state transition matrix (STM) captures the history of state transitions. Each state is a range of CPU utilization percentages; with state resolution (r) of 5%, there are a total of $\frac{100}{5} = 20$ states. The current state is determined based on the latest utilization observation: $i = \lceil 100\%/x_n \rceil$, and the forecasted utilization is:

$$x_{n+1} = \frac{\sum_{j=1}^{\lceil 100\%/r \rceil} STM_{i,j} \times (j+o)}{\sum_{j=1}^{\lceil 100\%/r \rceil} STM_{i,j}} \times (1+m) \quad (5)$$

Here, o is an offset to compensate quantization of values.

Other predictors: We also implemented Double Exponential Smoothing, Autoregressive, Passive Aggressive Regression, and ARIMA predictors. We do not present their description and results for brevity due to their mediocre performance compared to SES and SMA despite more complexity.

Figure 12 compares prediction scores for these four policies. It also shows an Oracle policy where future is known, in which case RE is always 100% and V is always 0%. We only include VMs with a minimum lifetime of 30 minutes (65% of all VMs), as 5-minute readings mean that shorter lifetimes require 4 or fewer predictions; too few to draw meaningful statistical conclusions from. We see that any active prediction policy is significantly better than relying on the latest observation of peak utilization. While we picked MC for delivering slightly better scores, we do not observe considerable differences among these policies. To better understand what limits prediction scores, we look at resource reclamation efficiency and prediction violations of the MC prediction policy separately, as a function of VM lifetime (Figure 13). Results here include any VM with lifetime of at least 15 minutes (91%

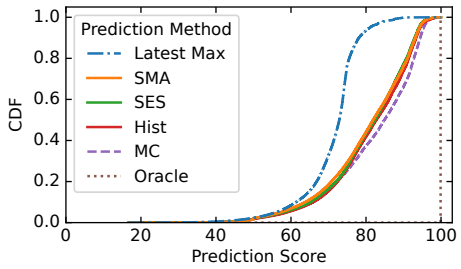


Figure 12: Prediction scores for simulated policies.

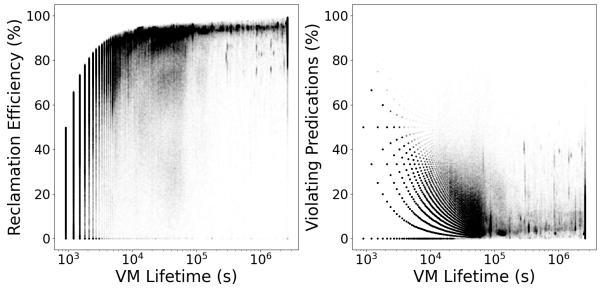


Figure 13: Resource reclamation efficiency and prediction violations as a function of VM lifetime for 1 million VMs.

of all 1 million VMs). We observe that with increased VM lifetime, the resource efficiency is increased and violations are reduced. This is not surprising, as with longer history, policy parameters can be better trained for each VM.

10 Related Work

Table 2 compares UnFaaSener with the related works targeting offloading in the serverless domain. For a category of related work, the application is primarily deployed on VMs and serverless plays the backup role at scale-out, while new VMs are being provisioned [43, 47, 63, 84, 87]. LIBRA [67] extends these works by simultaneously utilizing FaaS for the low-rate bursty portion of the traffic. Reliance on VMs as the primary infrastructure limits the scope of such systems to specific domains (as seen in Table 2), and in effect distances them from serverless's high scalability and pay-per-use features.

Another category of work proposes building hybrid IaaS-FaaS deployments [56, 75]. The main drawback of these systems is adding a second scheduling/control layer on top of that of serverless platforms. Moving away from the scheduler of public serverless offerings as the primary scheduler and adding a new layer limits the scalability of these approaches, and comes with reliability and security implications. UnFaaSener relies on the serverless scheduler to ensure a secure and scalable gateway to external events, and by using pre-solved decisions, eliminates added scheduling overheads.

Lastly, a category of work proposes modifying the serverless platform to enable serverless functions to use resource-harvesting VMs [86], idle resources from over-allocated serverless functions [83], and users' VMs on the same platform [78]. These proposals modify cloud providers extensively, which is out of reach of end users, and limit offloading to the scheduler's scope, usually within the same cluster. UnFaaSener works on top of existing serverless systems, requires no change to the platform, and puts no limit on the location of hosts it is harvesting resources from.

Related Work	Supports Complex DAGs	Scheduling/ Control Path	General Purpose	Resource Harvesting	Primary Deployment	Partial Offloading Decisions
Splice [75]	✗	FaaS scheduler + Custom Scheduler	✓	✗	Hybrid	✗
Spock [43]	✗	FaaS scheduler + Custom Scheduler	✗ (ML Inference)	✗	IaaS	NA
SplitServe [47]	✗	FaaS scheduler + Custom Scheduler	✗ (Spark Jobs)	✗	IaaS	NA
MArk [84]	✗	FaaS scheduler + Greedy Instance Plan	✗ (ML Inference)	✗	IaaS	NA
Amoeba [56]	✗	FaaS scheduler + Custom Controller	✗ (Microservices)	✗	Hybrid	✗
FEAT [63]	✗	FaaS scheduler + Custom Controller	✓	✗	IaaS	NA
LIBRA [67]	✗	FaaS scheduler + Custom Controller	✓	✗	IaaS	✓ (offloads excess traffic)
ServerMore [78]	✗	FaaS scheduler + Custom Controller	✓	✓	FaaS	✗
Skedulix [31]	✓	FaaS scheduler + Custom Scheduler	✓	✗	Private FaaS	✗
Kraken [20]	✓	Kraken Scheduler	✓	✗	FaaS	NA
Freyr [83]	✗	Serverless Controller + Resource Manager	✓	✓	FaaS	NA
Zhang et al. [86]	✗	FaaS scheduler	✓	✓	FaaS	NA
BeeHive [87]	✗	FaaS scheduler + Custom Runtime	✗ (Web Apps)	✗	IaaS	✓ (sets offloading ratio)
UnFaaSener	✓	FaaS scheduler + lightweight LUT	✓	✓	FaaS	✓

Table 2: The taxonomy of related work.

11 Discussion

Threat model. UnFaaSener offloads users' functions to their own offloading hosts. This simplifies the threat model by eliminating co-location of different teams' applications on the same host. Developers have full control: they specify offloading hosts, install host agents, and can unsubscribe hosts at any time. UnFaaSener's execution agent runs offloaded functions in separate Docker containers for isolation, but this also brings security implications [77]. In this context, we assume that 1) the host, which belongs to the same development team, is not malicious and 2) the developer is aware of the data protection implications of offloading functions.

12 Conclusion

UnFaaSener enables serverless developers to leverage the unused capacity of their VMs or on-premise servers, yielding substantial cost savings without modifying existing serverless platforms. UnFaaSener lays a foundation for researchers to explore the potential of the proposed serverless offloading mechanism for diverse purposes beyond cost optimization.

13 Acknowledgements

We thank members of the UBC CIRRUS Lab for their feedback on this work, and thank Ila Nimgaonkar for suggesting the name UnFaaSener. We also thank the anonymous reviewers and our shepherd, Chia-Che Tsai, for helping us improve the paper. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) grants RGPIN-2021-03714 and DGECR-2021-00462. This work was made possible by cloud resources from Digital Research Alliance of Canada, Google Cloud Research Credits program, and AWS Cloud Credit for Research program.

References

[1] pandas, Accessed: 2023-01-05. https://pandas.pydata.org/.

[2] AWS Lambda Power Tuning, Accessed on 2023-01-05. https://github.com/alexcasalboni/aws-lambda-power-tuning.

[3] AWS Lambda Pricing, Accessed on 2023-01-05. https://aws.amazon.com/lambda/pricing/.

[4] AWS Step Functions Pricing, Accessed on 2023-01-05. https://aws.amazon.com/step-functions/pricing/.

[5] AWS Step Functions: Visual workflows for distributed applications, Accessed on 2023-01-05. https://aws.amazon.com/step-functions/.

[6] Azure Functions Invocation Trace 2021, Accessed on 2023-01-05. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md.

[7] CloudSuite: Graph Analytics, Accessed on 2023-01-05. https://github.com/parsa-epfl/cloudsuite/blob/CSv3/docs/benchmarks/graph-analytics.md.

[8] Docker Hub: Build and ship any application anywhere, Accessed on 2023-01-05. https://hub.docker.com/.

[9] FaaSProfiler, Accessed on 2023-01-05. https://github.com/PrincetonUniversity/faas-profiler.

[10] Google Cloud Workflows, Accessed on 2023-01-05. https://cloud.google.com/workflows.

[11] Google text-to-speech API, Accessed on 2023-01-05. https://cloud.google.com/text-to-speech.

[12] A PaaS end-to-end ML setup with Metaflow, serverless and SageMaker., Accessed on 2023-01-05. https://github.com/jacopotagliabue/no-ops-machine-learning.

[13] vSwarm: A suite of representative serverless cloud-agnostic (i.e., dockerized) benchmarks, Accessed on 2023-01-05. https://github.com/vhive-serverless/vSwarm.

[14] What is Pub/Sub?, Accessed on 2023-01-05. https://cloud.google.com/pubsub/docs/overview.

[15] Workflows pricing, Accessed on 2023-01-05. https://cloud.google.com/workflows/pricing.

[16] Paarijaat Aditya, Istemi Ekin Akkus, Andre Beck, Ruichuan Chen, Volker Hilt, Ivica Rimac, Klaus Satzke, and Manuel Stein. Will serverless computing revolutionize NFV? *Proceedings of the IEEE*, 107(4):667–678, 2019.

[17] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[18] R. Allard. Use of time-series analysis in infectious disease surveillance. *Bulletin of the World Health Organization*, 76(4):327, 1998.

[19] Logan Beal, Daniel Hill, R Martin, and John Hedengren. GEKKO optimization suite. *Processes*, 6(8):106, 2018.

[20] 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*, SoCC '21, page 153–167. ACM, 2021.

[21] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 381–397. ACM, 2023.

[22] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

[23] A. Carpinone, M. Giorgio, R. Langella, and A. Testa. Markov chain modeling for very-short-term wind power forecasting. *Electric Power Systems Research*, 122:152–158, 2015.

[24] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[25] Francois Chollet et al. Keras, Accessed: 2022-10-13. https://github.com/fchollet/keras.

[26] Claudio Cicconetti, Marco Conti, Andrea Passarella, and Dario Sabella. Toward distributed computing environments with serverless solutions in edge systems. *IEEE Communications Magazine*, 58(3):40–46, 2020.

[27] Google Cloud. gcloud CLI overview, Accessed on 2023-01-05. https://cloud.google.com/sdk/gcloud.

[28] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78. ACM, 2021.

[29] Eli Cortez. Azure public dataset v1, Accessed on 2023-01-05. https://github.com/Azure/AzurePublicDataset/blob/master/AzurePublicDatasetV1.md.

[30] 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, page 153–167. ACM, 2017.

[31] Anirban Das, Andrew Leaf, Carlos A. Varela, and Stacy Patterson. Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 609–618, 2020.

[32] Datadog. The state of serverless, May 2021. https://www.datadoghq.com/state-of-serverless-2021/.

[33] Datadog. The state of serverless, June 2022. https://www.datadoghq.com/state-of-serverless/.

[34] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 265–276, April 2020.

[35] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering*, 2021.

[36] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2020.

[37] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[38] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[39] 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*, page 583–594. ACM, 2022.

[40] 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*, ASPLOS '21, page 386–400. ACM, 2021.

[41] Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149, 2022.

[42] Samuel Ginzburg and Michael J Freedman. Serverless isn't server-less: Measuring and exploiting resource variability on cloud FaaS platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 43–48, 2020.

[43] 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, 2019.

[44] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of

Alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19. ACM, 2019.

[45] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861. USENIX Association, November 2020.

[46] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262, 2018.

[47] Aman Jain, Ata F. Baarzi, George Kesidis, Bhuvan Urgaonkar, Nader Alfares, and Mahmut Kandemir. SplitServe: Efficiently splitting apache Spark jobs across faas and iaas. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 236–250. ACM, 2020.

[48] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cerin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing colocated workloads in Alibaba cloud datacenters. *IEEE Transactions on Cloud Computing*, 2020.

[49] Yuxuan Jiang, Mohammad Shahrad, David Wentzlaff, Danny HK Tsang, and Carlee Joe-Wong. Burstable instances for clouds: Performance modeling, equilibrium analysis, and revenue maximization. *IEEE/ACM Transactions on Networking*, 28(6):2489–2502, 2020.

[50] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: Principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 289–305. ACM, 2022.

[51] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[52] Haneul Ko, Sangheon Pack, and Victor C. M. Leung. Performance optimization of serverless computing for latency-guaranteed and energy-efficient task offloading in energy harvesting industrial IoT. *IEEE Internet of Things Journal*, 2021.

[53] Alexey Kopytov. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/*, 2004.

[54] Samuel Kounev, Cristina Abad, Ian T. Foster, Nikolas Herbst, Alexandru Iosup, Samer Al-Kiswany, Ahmed Ali-Eldin Hassan, Bartosz Balis, André Bauer, André B. Bondi, Kyle Chard, Ryan L. Chard, Robert Chatley, Andrew A. Chien, A. Jesse Jiryu Davis, Jesse Donkervliet, Simon Eismann, Erik Elmroth, Nicola Ferrier, Hans-Arno Jacobsen, Pooyan Jamshidi, Georgios Kousiouris, Philipp Leitner, Pedro Garcia Lopez, Martina Maggio, Maciej Malawski, Bernard Metzler, Vinod Muthusamy, Alessandro V. Papadopoulos, Panos Patros, Guillaume Pierre, Omer F. Rana, Robert P. Ricci, Joel Scheuner, Mina Sedaghat, Mohammad Shahrad, Prashant Shenoy, Josef Spillner, Davide Taibi, Douglas Thain, Animesh Trivedi, Alexandru Uta, Vincent van Beek, Erwin van Eyk, André van Hoorn, Soam Vasani, Florian Wamser, Guido Wirtz, and Vladimir Yussupov. Toward a Definition for Serverless Computing. In *Serverless Computing (Dagstuhl Seminar 21201)*, volume 11, pages 34–93. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021.

[55] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit A. Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. Prediction-Based power oversubscription in cloud platforms. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 473–487. USENIX Association, July 2021.

[56] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. Amoeba: QoS-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408, 2020.

[57] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, 54(10s), September 2022.

[58] Álvaro López García, Jesús Marco De Lucas, Marica Antonacci, Wolfgang Zu Castell, Mario David, Marcus Hardt, Lara Lloret Iglesias, Germán Moltó, Marcin Plociennik, Viet Tran, Andy S. Alic, Miguel Caballer, Isabel Campos Plasencia, Alessandro Costantini, Stefan Dlugolinsky, Doina Cristina Duma, Giacinto Donvito, Jorge Gomes, Ignacio Heredia Cacha, Keiichi Ito, Valentin Y. Kozlov, Giang Nguyen, Pablo Orviz Fernández, Zděnek Šustr, and Pawel Wolniewicz. A cloud-based framework for machine learning workloads and applications. *IEEE Access*, 8:18681–18692, 2020.

[59] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh

Bagchi, and Somali Chaterji. WiseFuse: Workload characterization and DAG transformation for serverless workflows. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

[60] Stefan Majiros. Nightly and serverless builds in MS appcenter for React native - async mobile DevOps example, 2021. https://stefan-majiros.com/blog/nightly-serverless-builds-with-app-center-for-react-native-async-mobile-devops/.

[61] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA, October 2018. USENIX Association.

[62] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: An opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 228–244. ACM, 2021.

[63] Joe H. Novak, Sneha Kumar Kasera, and Ryan Stutsman. Cloud functions for fast and robust resource auto-scaling. In *2019 11th International Conference on Communication Systems Networks (COMSNETS)*, pages 133–140, 2019.

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

[65] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *2016 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 122–132. IEEE, 2016.

[66] Haoran Qiu, Saurabh Jha, Subho S. Banerjee, Archit Patke, Chen Wang, Franke Hubertus, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Is function-as-a-service a good fit for latency-critical services? In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, WoSC '21, page 1–8. ACM, 2021.

[67] Ali Raza, Zongshun Zhang, Nabeel Akhtar, Vatche Isahagian, and Ibrahim Matta. LIBRA: An economical hybrid approach for cloud applications with strict SLAs. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 136–146, 2021.

[68] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 122–137. ACM, 2021.

[69] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 753–767. ACM, 2022.

[70] Marco Savi, Alessandro Banfi, Alessandro Tundo, and Michele Ciavotta. Serverless computing for NFV: Is it worth it? a performance comparison analysis. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 680–685, 2022.

[71] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1063–1075. ACM, 2019.

[72] 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 *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[73] 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*, SoCC '20, page 281–295. ACM, 2020.

[74] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. ACM, 2019.

[75] Myungjun Son, Shruti Mohanty, Jashwant Raj Gunasekaran, Aman Jain, Mahmut Taylan Kandemir, George Kesidis, and Bhuvan Urgaonkar. Splice: An automated framework for cost-and performance-aware blending of cloud services. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 119–128, 2022.

[76] Chris Spencer. CriticalPath Python package, Accessed: 2023-01-05. https://pypi.org/project/criticalpath/.

[77] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2019.

[78] Amoghavarsha Suresh and Anshul Gandhi. Servermore: Opportunistic execution of serverless functions in the cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 570–584. ACM, 2021.

[79] Ajay Tirumala. Iperf: The TCP/UDP bandwidth measurement tool. *http://dast. nlanr. net/Projects/Iperf/*, 1999.

[80] 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, page 1–16. ACM, 2021.

[81] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: A native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 768–781. ACM, 2022.

[82] 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*, SEC '17. ACM, 2017.

[83] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. Accelerating serverless computing by harvesting idle resources. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 1741–1751. ACM, 2022.

[84] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[85] Lu Zhang, Weiqi Feng, Chao Li, Xiaofeng Hou, Pengyu Wang, Jing Wang, and Minyi Guo. Tapping into NFV environment for opportunistic serverless edge function deployment. *IEEE Transactions on Computers*, 71(10):2698–2704, 2022.

[86] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739. ACM, 2021.

[87] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. BeeHive: Sub-second elasticity for web services with semi-FaaS execution. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 74–87. ACM, 2023.

A Appendix

A.1 Dynamic Concurrency Feedback Details

As mentioned in §7.3, the execution agent uses a feedback mechanism to dynamically set the concurrency limit. Execution time changes during the time (ΔExecTime), which is used as feedback in Equation 2, is calculated as:

$\overline{E_n}^t \leftarrow$ Current Average Execution Time for Function$_n$
${E_n}^t \leftarrow$ Current Execution Time for Function$_n$
$\overline{E_n}^{t+1} = 0.8 \times \overline{E_n}^t + 0.2 \times {E_n}^t$
$\Delta {E_n}^t = \overline{E_n}^t - {E_n}^{t-1}$
$\Delta\text{ExecTime} = \sum_{n \in \text{functions}} \Delta {E_n}^t$

In order to prevent frequent changes in the concurrency limit, we quantized the value of the μ calculated by Equation 2 into $[0.33, 0.66, 1]$. The weight of the exponential term in Equation 2 (0.03) was determined empirically, considering this quantization. Figure 14 shows the feedback mechanism used for setting the concurrency limit.

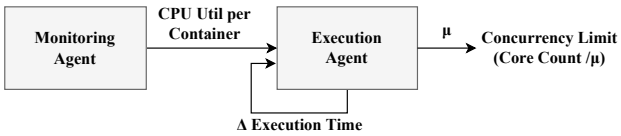


Figure 14: The feedback mechanism for dynamic concurrency.

A.2 Sample DAG JSON description file

```
{
"workflow": "Text2SpeechWorkflow",
"workflowFunctions": [
  "Text2SpeechWorkflow_GetInput",
  "Text2SpeechWorkflow_TransferInput",
  "Text2SpeechWorkflow_Profanity",
  "Text2SpeechWorkflow_Text2Speech",
  "Text2SpeechWorkflow_Conversion",
  "Text2SpeechWorkflow_Compression",
  "Text2SpeechWorkflow_MergeFunction",
  "Text2SpeechWorkflow_Censor"
],
"initFunc":"Text2SpeechWorkflow_GetInput",
"successors": [
  ["Text2SpeechWorkflow_TransferInput"],
  ["Text2SpeechWorkflow_Profanity",
  "Text2SpeechWorkflow_Text2Speech"],
  ["Text2SpeechWorkflow_MergeFunction"],
  ["Text2SpeechWorkflow_Conversion"],
  ["Text2SpeechWorkflow_Compression"],
  ["Text2SpeechWorkflow_MergeFunction"],
  ["Text2SpeechWorkflow_Censor"],
  []
],
"predecessors": [
  [],
  ["Text2SpeechWorkflow_GetInput"],
  ["Text2SpeechWorkflow_TransferInput"],
  ["Text2SpeechWorkflow_TransferInput"],
  ["Text2SpeechWorkflow_Text2Speech"],
  ["Text2SpeechWorkflow_Conversion"],
  ["Text2SpeechWorkflow_Compression",
  "Text2SpeechWorkflow_Profanity"],
  ["Text2SpeechWorkflow_MergeFunction"]
]
}
```

Figure 15: A sample DAG JSON description file.

LLFree: Scalable and Optionally-Persistent Page-Frame Allocation

Lars Wrenger
Leibniz Universität Hannover

Florian Rommel
Leibniz Universität Hannover

Alexander Halbuer
Leibniz Universität Hannover

Christian Dietrich
Hamburg University of Technology

Daniel Lohmann
Leibniz Universität Hannover

Abstract

Within the operating-system's memory-management subsystem, the page-frame allocator is the most fundamental component. It administers the physical-memory frames, which are required to populate the page-table tree. Although the appearance of heterogeneous, nonvolatile, and huge memories has drastically changed the memory hierarchy, we still manage our physical memory with the seminal methods from the 1960s.

With this paper, we argue that it is time to revisit the design of page-frame allocators. We demonstrate that the Linux frame allocator not only scales poorly on multi-core systems, but it also comes with a high memory overhead, suffers from huge-frame fragmentation, and uses scattered data structures that hinder its usage as a persistent-memory allocator. With LLFree, we provide a new lock- and log-free allocator design that scales well, has a small memory footprint, and is readily applicable to nonvolatile memory. LLFree uses cache-friendly data structures and exhibits antifragmentation behavior without inducing additional performance overheads. Compared to the Linux frame allocator, LLFree reduces the allocation time for concurrent 4 KiB allocations by up to 88 percent and for 2 MiB allocations by up to 98 percent. For memory compaction, LLFree decreases the number of required page movements by 64 percent.

1 Introduction

In any virtual-memory subsystem, the allocation of *physical memory* is a vital base primitive. Classically, the OS hands out physical memory in *page frames* of MMU-imposed sizes and uses simple free lists [42, 50] (Windows, Darwin) or specialized buddy-allocators [28] (Linux, FreeBSD) to manage multiple frame sizes. However, recent hardware trends (i.e., high core counts and NVRAM) challenge these *frame-allocator* designs.

One significant trend is the appearance of fast [49] byte-addressable *nonvolatile RAM (NVRAM)* in the form of Intel Optane DIMMs. Inspired by their persistence property, new programming models [5, 39, 45, 52], file systems [10, 40, 48], and crash-tolerant data structures [8, 13, 46] were proposed and evaluated. And while Intel's announcement [25] to wind down its Optane business will make NVRAM harder to obtain in the next years, it has been shown that low-cost, high-capacity NVRAM is feasible and has significant potential [1, 23, 31]. Furthermore, multiple researchers realized that persistence and scalability are deeply entangled properties [26, 29] that benefit both from lock-free algorithms [8, 16, 46] and constructive avoidance of inconsistent intermediate states.

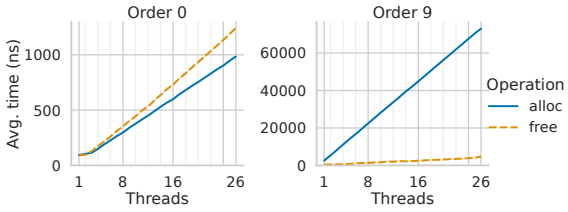


Figure 1: Linux frame allocator performance for concurrent allocations of 4 KiB (order 0) and 2 MiB (order 9) huge frames.

Together with many cores competing for resources, large-capacity NVRAM modules provoke the question of how, at which costs, and with what guarantees the OS hands out the available memory, which might be used for persistent data. For example, for databases, current virtual-memory subsystems can have a significant impact on their design [12, 35], query-processing speeds [14] and buffer management [12, 32]. Therefore, we believe it is time to revisit the whole virtual-memory stack, starting from the bottom; the frame allocator.

First, we investigated whether the Linux frame allocator [18] and its underlying buddy system [28] still match today's requirements, not only for safely allocating frames from NVRAM but also for the scalable management of DRAM. Fig. 1 shows the multi-core scalability of bulk allocations. With all 26 cores allocating in parallel, 4 KiB allocations slow down by a factor of *ten* (94 ns→984 ns), while 2 MiB alloca-

tions are even 27 times slower! This poor scalability affects many multicore and memory-heavy workloads [7]. The root causes are the scattered allocator state and the usage of global locks, both of which are also problematic [16, 17] for a crash-tolerant NVRAM adaptation.

About this paper

We propose LLFREE, a persistent, lock- and log–free page-frame allocator that: (1) focuses on *memory-management unit (MMU)*-specific memory sizes, (2) scales well on multi-core systems by reducing memory sharing, (3) is memory efficient due to its small amount of metadata, (4) has automatic huge-frame defragmentation, and (5) is always in a consistent state and, thus, well suited for persistent memory. In this paper, we claim the following contributions:

- We explore the weaknesses of the Linux buddy allocator and simpler list-based frame allocators.
- We derive design principles for hardware-centric lock- and log-free physical memory management.
- With LLFREE, we provide a page-frame allocator that is suited for both volatile and nonvolatile memory.
- We replace the Linux buddy allocator with LLFREE and conduct a comprehensive evaluation to compare the two allocators in terms of performance, scalability, spatial overhead, fragmentation behavior, and crash consistency.

2 Problem Analysis: Linux Frame Allocator

The page-frame allocator must provide physical-memory *frames*, which have MMU-specific granularities, are naturally aligned, and are used to set up virtual address spaces. For this paper, we will stick to the AMD64 MMU and its frame sizes (4 KiB, 2 MiB, 1 GiB), which we call *natural (allocation) sizes*. However, the general design can be adapted to other page sizes. For 4 KiB, we will use the term *base frame*, for 2 MiB *huge frame*, and for 1 GiB *giant frame*, respectively. While some kernels (e.g., Windows and Darwin) use simple free lists [42, 50], Linux (and FreeBSD) use the buddy system [18, 28].

Linux Buddy Allocator A buddy allocator avoids external fragmentation by allowing only allocation sizes of the form $2^o \times P$, where P is the smallest size and o is the allocation *order*. For each order, the buddy system keeps a bucket of naturally-aligned free blocks. If an allocation hits an empty bucket o, a block from the bucket $o+1$ is requested and halved into two *buddies*, whose start addresses differ only in a single bit. One buddy is returned, the other is put into the order-o bucket. The free operation tries to recursively merge the block with already freed buddy blocks before putting the block into a bucket. To speed up merging, buckets are usually implemented as doubly-linked lists and a one-bit flag is used to track which blocks are available for merging.

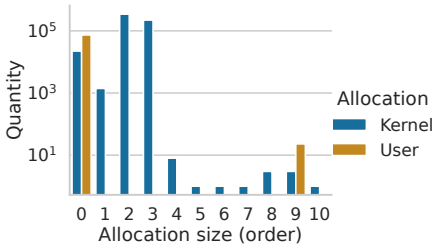


Figure 2: Requested allocation sizes during system startup and a 120 s memcached+memtier benchmark.

Linux employs one buddy allocator per memory zone (e.g., for each NUMA node), supports the orders $o \in \{0, \ldots, 10\}$, and uses the base-frame size as P. Therefore, the supported sizes are between 4 KiB and 4 MiB on AMD64. Unlike a general-purpose buddy allocator, Linux does not store the list pointers within the free memory, as this would require all memory to be mapped, which is not supported by all architectures. Instead, it uses the `struct page` for its metadata.[1] Also, each zone allocator is protected by a spin lock, which serializes all split and merge operations. In order to reduce contention on these locks, Linux further employs per-CPU caches for the most frequently-used orders.

Problem 1: Mixing of Concerns Linux's frame allocator is not only used for hardware-sized page frames but *also* for allocating contiguous physical-memory ranges of various orders. Although this was necessary for allocating *direct-memory access (DMA)* buffers before the widespread adoption of I/O-MMUs, there is no technical requirement for this anymore. However, the Linux developers still use those non-native sizes to allocate larger kernel objects (e.g., stacks). To get a feeling for this, we recorded the requested sizes during boot and a subsequent `memcached` benchmark (see Fig. 2): We see that userspace memory gets only requested for the natural orders 0 and 9, whereas the kernel mostly requests non-natural orders. Allocating contiguous blocks of frames for kernel objects might still be beneficial to save TLB entries (Linux identity-maps all physical memory into the kernel space with giant frames). However, by mixing frame- and kernel-object allocation, the allocator has to provide all orders via its interface, which leads to a number of secondary problems.

Problem 2: Merge Cost Because there are nine buddy orders between the base and huge frames, the transition between both is costly: In the worst case, starting from 512 4 KiB frames, it takes 511 buddy-merge operations to form a single 2 MiB frame. For each merge and under lock, we have to manipulate list pointers in five cache lines; four are in `struct pages` that are usually not yet in the cache.

Problem 3: Scalability Furthermore, as we have seen in Fig. 1, these already costly operations scale poorly if re-

[1] With `struct page`, Linux has a per-frame information store that is used and repurposed by different subsystems. On AMD64, it is 64 bytes large.

quested by multiple threads. The reason for this is the contention at the mentioned per-zone lock, which Linux tries to mitigate by maintaining per-CPU caches for some orders. Each per-CPU cache keeps a list of free blocks, which are drained on memory pressure or if the cache exceeds a watermark. While for a long time, only order-0 allocations were cached, Linux 5.13 extended the caches to order 1-3 and order 9 (2 MiB). However, allocation-heavy workloads easily overwhelm these per-CPU caches. Also, they aggravate fragmentation and complexity.

Problem 4: Huge-Frame Fragmentation Although the buddy system prioritizes the smallest-fitting bucket, it has a problem with huge-page fragmentation: For example, if a single 4 KiB piece of an otherwise free 2 MiB frame is in use, the other 511 base frames exist as 9 blocks of different orders (4 KiB to 1 MiB). Since buckets are unsorted sets and the allocator has no concept of "almost-full huge pages", those blocks have equal chances of being allocated as any other block from any other huge frame. As a result, the buddy system does not specifically aim to minimize the fragmentation of huge frames. We will discuss this further in Sec. 5.6.

The per-CPU caches aggravate fragmentation as they delay merge operations. Even if the last missing 4 KiB frame of a 2 MiB frame has already been freed, it may reside in a per-CPU cache that would have to be flushed to complete the huge frame. Furthermore, as the per-CPU cache hides memory from the buddy allocator, we cannot employ an intra-bucket heuristic that increases the likelihood of a huge-frame merge (e.g., appending to the end of the bucket list). On the contrary, a recently freed block that *could* complete a 2 MiB frame is more likely to be allocated again.

In order to reduce huge-page fragmentation, Linux has supported "high-atomic page blocks" since 2015 to isolate larger blocks and prevent them from being fragmented by small allocations. Furthermore, Linux also employs active defragmentation (memory compaction), where a background task iterates through a memory zone and moves pages to the beginning, clearing larger chunks at the end. However, both induce additional complexity.

Problem 5: Persistent Allocations Given its current structure, the Linux frame allocator is unsuitable for persistent allocations. For persistent NVRAM zones, the allocator must be able to recover its state in case of a power loss to ensure the persistence of the required metadata. This is challenging [17] for complex algorithms (such as lock-protected recursive frame merging), distributed state (such as doubly linked lists), or redundancy (such as per-CPU caches). While in theory, each of these problems could be solved with extra logging protocols [36, 41, 45], the performance, memory, and complexity impact of doing so would be excessive. Also, this logging overhead would have to be paid for every regular operation despite crashes being usually extremely rare. Thus, we do not consider this as a realistic option. To our knowledge, there are currently no persistent (page-frame) allocators that achieve allocation times close to their volatile counterparts [3, 9, 33, 36, 41, 45, 52].

Problem Summary: Complexity In the end, the Linux physical-memory allocator suffers from complexity. Mixing the concerns of frame-sized and other allocation quantities (Problem 1) motivates the buddy structure, which, however, leads to high merge costs (Problem 2) and lock-based, doubly-linked traversals that hamper multi-core scalability (Problem 3). This is mitigated by an increasing number of per-CPU caches, which (combined with the buddy system) unfortunately worsens huge-frame fragmentation (Problem 4), requiring additional mechanisms such as high-atomic page blocks and memory compaction that further increase the complexity. All this results in a design that is unsuitable for persistent allocations (Problem 5) due to redundant and distributed storage of data and state.

All these design decisions were most probably well-founded when they were made. We argue that the time has come to revisit the design and structure of the most fundamental memory manager in our systems, the page-frame allocator.

3 The LLFree Page-Frame Allocator

We originally designed LLFree as a page-frame allocator for natural frame sizes (4 KiB, 2 MiB) only, with the goal of high scalability and suitability for persistent allocations. For the Linux integration, we later had to extend it to support also non-natural allocation orders, which to our surprise, worked out without having to compromise on any of our design goals. In the following, we describe the original design, while the integration particularities are left to Sec. 4.

3.1 Design Principles

For our allocator, we specified three major design principles:

Respect Hardware The hardware characteristics define both the structural elements for and the features of the software implementation. We leverage MMU-defined frame sizes, cache-line granularity in data structures, and available atomics in algorithms.

Avoid Sharing On systems with multiple CPUs, both true and false sharing are major bottlenecks for scalability, becoming even more significant on NUMA systems. Locks are a frequent cause of sharing. We reduce access to shared data structures and do not use locks.

Careful Redundancy Redundant information (such as software caches and replicas) must be kept in sync. This is especially difficult to accomplish when targeting crash consistency on persistent memory since a potential crash can disrupt the synchronization of these redundant data structures. Therefore, we strictly limit redundancy for the state that is required for crash recovery.

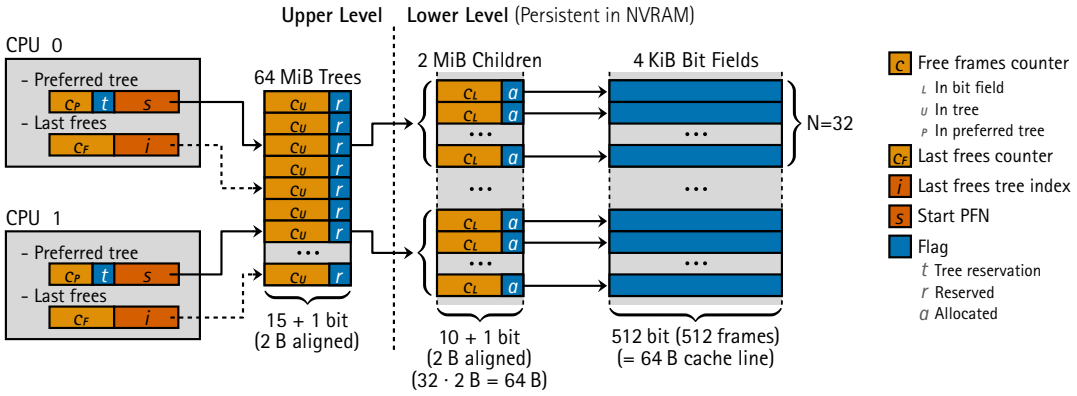


Figure 3: Architecture of the LLFREE allocator: The tree entries, child entries, and bit fields are stored consecutively in large arrays. From a PFN, we can directly extract the indices of the corresponding tree entry, child entry, bit field, and the frame's allocated bit within.

We do not claim fundamental novelty for these principles: The first two are well established in the domain of scalable OS kernel development; their benefits have been reported many times [6, 15, 22, 47]. This does not particularly hold for the third principle, which is more targeted at reaching crash consistency than scalability. On the contrary, employing rather than limiting redundancy is a common technique in kernel design to avoid sharing and *improve* scalability (as we have seen in the previous section), so there is a trade-off between the two. This deep entangling of persistence and scalability, as well as the importance of avoiding intermediate states for crash consistency on NVRAM, has already been reported in the domain of data structures and algorithms [8, 16, 26, 29, 46]. The contribution of this paper is the rigorous combination and application of these principles and dealing with their trade-offs in the design of a page-frame allocator that is *both* scalable on DRAM *and* optionally crash-consistent on NVRAM.

3.2 LLFREE: Design Overview

Fig. 3 depicts the architecture of LLFREE, which has been designed for the natural allocation orders 0 (4 KiB base frames) and 9 (2 MiB huge frames). Conceptually, LLFREE is divided into two levels: A *lower level* that performs the actual allocations and an *upper level* that implements allocation strategies to avoid sharing and fragmentation. Roughly speaking, the lower level takes responsibility for crash consistency – only its state needs to be kept persistent on a crash-consistent NVRAM zone – while the upper level provides for scalability.

3.2.1 Lower Level - Allocation Mechanisms

The lower level manages de/allocation of base and huge frames. For this, it employs a table entry and a bit field of 512 bits (Fig. 3: 2 MiB Children, 4 KiB Bit Fields) for every huge frame to mark the free (0) and taken (1) base frames from this huge frame. The number of free base frames is also maintained in the counter c_L. Base frames are allocated by first atomically decrementing c_L (this prevents races for the last free frame) and then searching the bit field for a 0 bit, an operation supported by special processor instructions. Huge frames are allocated by just changing the counter c_L from 512 to 0 and setting the *allocated flat* flag a, all within a single 16-bit *compare-and-swap (CAS)* operation. The bit field remains untouched, and all-zero in this case, the necessary bookkeeping (e.g., for crash recovery) is done in the a flag.

Thus, in most cases, the lower-level allocator needs to touch only two cache lines for de/allocating a base frame (table entry, bit field) and just one cache line for a huge frame (table entry).[2] Still, even if the current child/tree does not contain enough frames for an allocation, our sequential search fits well with the processor's cache-line prefetching.

3.2.2 Upper Level - Allocation Strategies

The upper level provides for scalability by using allocation strategies that constructively minimize (false) sharing and huge-frame fragmentation. Technically, it manages the physical memory as an array of chunks we call *trees* (Fig. 3): Each tree root refers to a lower-level table with N children that, in turn, refer to N bit fields, each managing 512 frames, similar to a page-table tree. We chose $N = 32$ so that child arrays (in the lower level) occupy a single cache line; hence a tree manages 16 384 base frames, respectively 64 MiB.

Each tree root contains the count c_U of free base frames (for allocation strategies), as well as a reserved flag r (for per-CPU pinning). Our early benchmarks showed that allocations suffer from false sharing when entries in the lower-level child array are updated concurrently. Hence, to avoid sharing, each CPU can pin ($r = 1$) a preferred tree for its allocations.

If there are no free frames left in the preferred tree, a new tree has to be reserved. The reservation algorithm follows a search heuristic to avoid fragmentation of huge frames by using c_U to classify trees into one of three classes:

[2] One additional cache-line access will happen in the upper level.

allocated	Almost all frames are taken ($c_U < 12.5\,\%$).
free	Almost all frames are free ($c_U > 87.5\,\%$).
partial	Everything in between.

The heuristic prioritizes *partial* trees over *free* and *allocated* ones. Thereby, (almost) *free* trees have a higher chance of becoming entirely free over time. However, we still select *free* trees before *allocated* ones for a new CPU-preferred tree, as the latter bear a high probability that allocations might (soon) fail again, especially for huge frames. This is a trade-off between performance and fragmentation.

The thresholds determining whether a tree is *free*, *partial*, or *allocated* are configurable. Our benchmarks showed that the thresholds of 12.5 percent (2048 free base frames for $N = 32$) for *free* and 87.5 percent for *allocated* ones are sufficient to avoid fragmentation (Sec. 5.6). The actual search for an appropriate tree is done in the following order:

1. First, the *neighborhood* of the current CPU tree is searched for *partial* or *free* trees, with *neighborhood* being defined as the 31 other tree roots that reside on the same cache line.
2. If this does not succeed, the whole trees array is sequentially searched (first-fit) for a *partial* tree.
3. If no *partial* tree is found, the search is repeated for a *free* tree or an *allocated* one with enough free pages.
4. As last resort, the allocator drains (unreserves) the trees of other CPUs and steals them.

Note, however, that even the slow path of reserving a new tree does not require locks and can be done fully parallel, as the reservation itself requires only the atomic update of the reserved flag of an entry. Given a memory zone of 256 GiB, the algorithm accesses 128 cache lines in the worst case.

CPU-Local Tree Roots: Despite using per-CPU reservations, the updates on the tree array can still suffer from false sharing when multiple CPUs concurrently update the free counters of entries sharing a cache line. This can be solved in two ways: either by aligning the 2 B tree entries to the cache line size (64 B on x86) or by splitting the counter into a global and a CPU-local part. We followed the second approach to keep the memory and cache overhead low. On tree reservation, a CPU moves the free counter c_U of the reserved tree to its CPU-local data c_P and sets c_U to zero. Allocations and frees from this CPU now change only the local counter c_P. Allocations from other CPUs no longer happen, as the tree is reserved, but foreign frees from previous allocations may still happen. In this case, only the global counter c_U in the trees array is incremented, which avoids invalidating the cache line of the respective local entry. The counters are synchronized if a local allocation runs out of memory ($c_P = 0$) or, in rare cases, by remotely draining a reserved tree from another CPU, which also clears the tree's reservation.

Besides the local free counter c_P, the CPU-local entries also contain an *in-tree reservation* flag t, and the *start PFN*, combined into a single 64 bit data type. The *start PFN* contains the last allocated base frame number; it acts as a last-fit pointer to speed up allocations in the child array (divide by 512) and also identifies the reserved tree (divide by $512 \cdot N$).

The t flag is atomically set during the reservation of a new per-CPU tree to prevent races with remote draining or parallel reservation attempts. The latter could only happen if the zone is too small for one tree per core. If this is the case, the per-CPU data is shared between multiple cores and the t-flag coordinates the allocation of a new tree.[3]

Reserve-on-Free: While per-CPU trees prevent contention and false sharing for concurrent allocations, frees must always go to the tree that was the origin of the respective frame. If the source CPU of the frame has meanwhile switched to a new tree or the free is invoked from another CPU, this tree is not the per-CPU tree. Especially on memory-intensive workloads, frees thereby may still suffer from false sharing.

To mitigate this, LLFREE provides a *reserve-on-free* heuristic that assumes that allocation and free workloads exhibit locality: A CPU reserves a tree as its preferred tree after F consecutive free operations targeted it, expecting that subsequent frees will also affect this tree (c_F and i in Fig. 3). The threshold $F = 4$ performed best in our benchmarks.

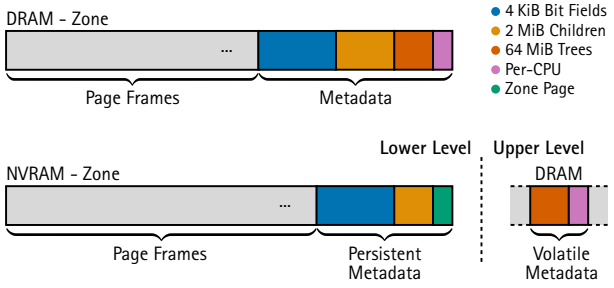


Figure 4: Layout of LLFREE-managed memory zones in persistent and non-persistent memory.

3.2.3 Crash Consistency

LLFREE optionally provides crash-consistency on NVRAM zones. For a crash-consistent NVRAM zone, only the allocator's lower level (cf. Fig. 3) has to be stored within the persistent memory – plus one extra zone page storing a magic identifier, the size of the memory region, and a startup/shutdown marker to detect if the system needs to be recovered after a power loss. The upper level always resides in DRAM to reduce access latencies and avoid unnecessary writes to NVRAM. It is restored into DRAM from the lower-level information. Fig. 4 depicts the zone layout for persistent and non-persistent memory zones.

[3] On Linux-x86, this holds only for the 16 MiB DMA zone, which Linux still offers for compatibility with legacy 16-bit ISA devices.

The key point of LLFree's design is that all transactions authoritatively happen within a single atomic change of a single cache line. For base frames, this is the atomic change in the respective bit field, while huge-frame de/allocation is implemented by the atomic change of a and c_L in the child entry. This ensures *memory consistency* as each authoritative change is visible to all cache-coherent cores, and all derived information (i.e., counter values) are altered by atomic commutative operations, which are not affected by reordering.

The single–cache-line property also makes it easy to implement *persistence consistency* on all systems that provide a *persist granularity* [38] of at least one cache line to be written atomically to the NVRAM at the end of the operation, which is considered to be the minimum standard for NVRAM hardware [10, 38]. In case of a recovery, each entry in the child array is checked for the a flag: If it is set, the child entry is the authoritative information – the entry is a huge page with $c_L = 0$ (the bit field is all-zero). Otherwise, the entry describes a set of base pages – the bit field is the authoritative information, which is used to restore the value of c_L. The upper-level state can then be restored from the child array.

While a crash during a de/allocation would never result in an unrecoverable allocator state, it could lead to a lost frame. This would happen if in an allocation the bit has already been set, but the frame has not yet reached the caller, or if the deallocation has been invoked but not yet cleared the bit. The theoretical worst-case bound for this effect is the maximum number of parallel operations, that is, the number of CPUs. This could be mitigated by two-phase de/allocation protocols, which, however, would also change the interface of the page allocator. Crashes are rare events – and a crash during the critical phase of a page-frame allocation even more so (the endangered code sequences take only a few clock cycles). Hence, the probability of an actual frame loss during the lifetime of a system is extremely low, while even in the case of such an incident, the costs would be acceptable.

4 Implementation

We implemented LLFree as a Rust module, integrated it with (and extended it for) the Linux kernel, and replaced the original Linux buddy allocator with LLFree. Its allocate and free algorithms, discussed in the previous section, can be found in the appendix (Fig. 14).

4.1 Approach

As the Linux community has started to adopt the Rust language, we took the chance to explore how well it is suited for a performance-critical low-level kernel module. Compared to C, Rust has much more restricted (i.e., safer) memory management and avoids undefined behavior. We are convinced that these properties, which prevent entire classes of memory bugs, simplified the development of the allocator.

The modular LLFree implementation contains a test environment that initializes the allocator on a virtual memory mapping in user space for unit testing and benchmarking. This made it possible to profile and find performance bottlenecks early on. Besides standard unit tests, we developed specific race-condition tests for the possible orders of atomic operations, which proved quite helpful in finding several design and logic errors.

Following this approach, we were able to quickly implement and compare strategies for the upper and lower level of the allocator. The final implementation consists of 2 199 lines of safe Rust code (with 25 unsafe lines for initialization and address translations) and 1 318 lines of unit tests. The Linux buddy allocator is written in C and mainly contained in `page_alloc.c` (excluding reclaiming and memory compaction), which alone has 6 060 lines of code – without any tests. However, the buddy allocator is tightly coupled with other memory subsystem components, making it difficult to estimate its actual contribution to this source base.

4.2 Replacing the Linux Buddy Allocator

We modified Linux to boot with our LLFree allocator. The integration required some modifications to LLFree (support for non-natural orders), but especially to Linux itself due to the tightly coupled implementation of the buddy allocator. Nonetheless, the system seems stable for everyday workloads.

4.2.1 LLFree Changes

LLFree was specifically designed for the natural orders defined by the hardware. Linux, however, requires supporting all orders up to 10, which we implemented as follows: Orders 1 to 6 (2 to 64 frames) can be allocated similarly to order 0. The allocation now searches for a large enough, aligned set of zeros in the bit fields and allocates it, toggling the bits with a single 64-bit CAS operation.

Orders 7 and 8 (128 and 256 frames) are allocated with an optimistic lock-free algorithm using 2–4 atomic operations. If one fails due to a race, the others are safely reverted, and the search continues. However, these orders are rarely allocated (cf. Fig. 2), and contention could only occur if a tree is stolen during an allocation or we explicitly share a tree among multiple CPUs. Hence, an actual conflict is a rare event.[4]

Order 10 was implemented by allocating two aligned child entries at once, similar to an order 9 allocation. As these child entries are only 16 bit large (with alignment), this is done with a single 32-bit CAS operation. If a higher-order allocation fails because the tree is fragmented, another one is reserved. In this case, the allocator searches for a *free* tree that is not fragmented before falling back on *partial* ones.

[4] This also breaks our assumption for persistence consistency from Sec. 3.2.3 for orders 7 and 8. However, we can safely ignore this for now, as non-natural orders are only employed by the kernel, which currently does not use persistence.

4.2.2 Linux Changes

During boot, the data structures for the LLFREE allocator are allocated by the early-boot `memblock` allocator and initialized. Like the buddy allocator, we create one LLFREE instance per memory zone and store a pointer to its data directly in the `struct zone`. Most further changes in the code base are to conditionally disable and replace the buddy allocator, its per-CPU caches, zone locks, and high-atomic page blocks when our implementation is activated (via a Kconfig option). In total, the LLFREE module, a thin wrapper around the LLFREE allocator, added 942 lines. Outside this module, we changed 415 lines with 296 alone in `page_alloc.c`.

Most functionalities, including page reclamation, were easily adapted to the new allocator. However, some higher-level services that directly use the buddy allocator's internal data structures, e.g., its free lists, turned out impossible to adapt without completely rewriting them. Hence, we disabled them for both allocators in the benchmarks. First, this includes the memory fragmentation heuristic that decides whether active memory compaction should be executed. The heuristic uses the internal counters of the buddy-allocator's free lists. However, the need for active memory compaction is an exceptional state, only triggered when the allocator is highly fragmented with almost no huge pages left. It does not happen in our benchmarks. Instead, we compare fragmentation and compaction costs in Sec. 5.6. Second, we deactivated the out-of-memory (OOM) handler, as its checks directly rely on the internal free lists of the buddy allocator. Considering the complexity of the OOM procedure, we consider its redesign is outside the scope of this paper. Our benchmarks do not trigger OOM events for both allocators.

To make the higher order allocation speeds in Linux competitive, we had to implement two workarounds: The first reduces the number of write accesses to the `struct page` entries. As an aid for kernel-internal debugging (e.g., detecting double frees), Linux reinitializes the flags of *all* `struct pages` backing a higher order allocation, which is a costly and mostly unnecessary overhead. Linux also differentiates between standard allocations, which can be freed in parts, and *compound* allocations, which can only be freed at once. For *compound* allocations, all but the first `struct page` are marked as *tail* page. This scattered, redundant encoding of *compound* frames is costly, especially for huge frames. Modifying all 512 entries (i.e., 512 cache lines) for every de/allocation is detrimental to the overall performance, making it hard to distinguish and compare the allocation speeds of both allocators. Thus, we benchmarked standard allocations and disabled the flag reinitialization. The latter did not have any observable consequences.

The second bottleneck that affected especially LLFREE is the updating of the vmstat free-frames counter, which provides an estimate of the available free memory. This counter is updated for de/allocations that escape the per-CPU caches. To reduce congestion, each CPU has a local counter absorbing updates up to a certain threshold. However, its current value of 125 is too small for huge frames. We increased the threshold by 1024 (the size of the order 9 per-CPU caches). Because the LLFREE allocator does not use these caches, the global counter remains as accurate as with the buddy allocator using the original value, as the latter may hide as many frames in its per-CPU caches.

5 Evaluation

In our evaluation, we show that LLFREE scales well for different allocation patterns and sizes. We also look at the fragmentation behavior, quantify the memory overhead, and investigate crash recovery for the NVRAM case.

5.1 Evaluation Setup and Benchmarks

Our test system is a DELL PowerEdge R750 with two Gen 3 Intel(R) Xeon(R) Gold 5320 CPUs (2× 26 physical cores @ 2.20 GHz). Each of the two NUMA nodes has four 32 GiB DRAM DIMMs (total: 256 GiB DRAM) and four 128 GiB Optane Gen 2 DIMMs (total: 1 TiB NVRAM). We assume the eADR persistence guarantees offered by the Gen 3 Xeon Gold architecture [24] (memory consistency ↦ persistence consistency) and omit explicit persisting cache flushes (`clwb`) in the implementation, as this also provides for a more direct comparison of DRAM and NVRAM allocation speeds.[5] For stable results, we disable proactive memory zeroing (a recently introduced optional hardening feature) and hyper-threading, which yields similar general performance characteristics, with the exception that memory sharing is costlier between physical cores than between logical ones. We execute our benchmarks on the modified Linux 6.0 kernel with and without LLFREE. As Linux instantiates allocators per memory zone (e.g., NUMA-1-DRAM) that do not impact each other, we perform isolated tests with a single NUMA-node allocator.

As allocator performance is highly work-load specific, we created three synthetic benchmarks that cover a wide range of allocation patterns: (1) For the *bulk* benchmark, all cores allocate half of the available memory at once and directly free it up again; this process is repeated. (2) The *random* benchmark allocates all memory(similar to the bulk benchmark) and frees the frames in random order; we only measure free operations. (3) For the *repeat* benchmark, each core allocates and frees a single frame as fast as possible. *Repeat* has been deliberately constructed as a best-case scenario for the Linux buddy allocator, as it maximizes the expected benefit of the local per-CPU caches. Nevertheless, this is the least realistic scenario: Common is lazy allocation (due to demand paging) with bulk/random free (at program termination).

[5] Note that LLFREE could also work with weaker persistency (Sec. 3.2.3).

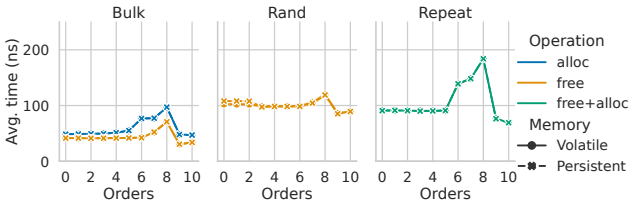


Figure 5: Per-order allocation time of standalone LLFREE on DRAM and NVRAM (8 cores, 128 GiB)

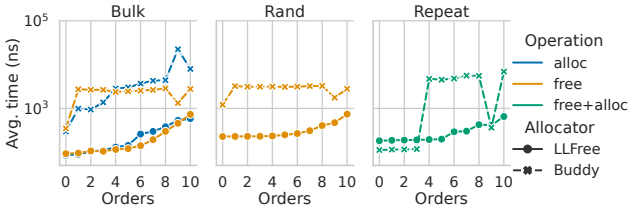


Figure 6: Per-order allocation time of Linux-integrated LLFREE (8 cores, 128 GiB DRAM, logarithmic scale)

As individual operations execute fast, per-operation time measurements would distort the results. Therefore, we measure the time for all operations and divide it by their number. Since LLFREE benefits from free locality, we expect the random benchmark to be especially challenging.

5.2 Allocation Sizes

First, we look at the allocation speeds of the different request sizes (4 KiB – 4 MiB). For this, eight CPU-pinned threads manage 128 GiB of DRAM or DAX-mapped Optane memory. The benchmarks are executed both in our userspace benchmark environment to measure the *standalone* performance of LLFREE and with a kernel module in the modified Linux.

Fig. 5 shows the average time per operation for the userspace benchmarks. For bulk and repeat, *one* operation costs less than 100 ns, while order 8 (1 MiB), which is the furthest from the next lower natural order, is the most expensive one. Due to random's cache-miss and invalidation behavior, a free operation can take up to 120 ns. Even though Optane is known to have around twice the random-access latency of DRAM [49], the resulting allocator performance is very similar for DRAM and NVRAM, as most updates remain in the L3 caches.

After these userspace results, we replaced the Linux buddy allocator with LLFREE for a quantitative in-situ comparison. This integration induces higher management overheads (e.g., updating `struct page`), which causes additional cache misses compared to the previous userspace benchmark. As Linux's allocator is not crash-consistent, we now only look at DRAM performance. Again, eight cores on the first NUMA node execute the respective benchmark in parallel.

Fig. 6 shows that LLFREE is about one order of magnitude faster than the original Linux allocator for the bulk and random benchmarks. For the repeat benchmark, where a single frame is reallocated repeatedly, the per-CPU caches (order 0-3, 9) make the buddy allocator faster than LLFREE (e.g., for order 0: 112 ns vs.183 ns). This stems not only from the caching itself but also from the fact that some statistics (i.e., `vmstat` counters, NUMA hit/miss rates) are not updated if the cache services a frame request. Nevertheless, for sizes that are not covered by per-CPU caches, LLFREE is about 100 times faster in the repeat benchmark.

However, the per-CPU caches are not beneficial for all workloads: In the bulk benchmark, we see that for order-9 allocations, the buddy allocator is about ten times slower than orders 8 and 10. An in-depth analysis revealed the problem: As the order-9 caches only have a capacity of two, the cache-refill operation is invoked for every other frame. This refill batches the allocation of multiple frames - two in this case - into a single critical section, reducing the number of acquire and release operations to the buddy lock. However, this critical section also contains a check for all `struct pages` of the allocated frames, which is especially expensive for higher-order allocations. Therefore, the lock is held longer, increasing lock contention compared to the other orders without caches that perform these checks after releasing the lock.

If we compare Fig. 5 and Fig. 6, we see that there is still potential for improvement in Linux's allocation path. As the other overheads scale linearly with the number of covered 4 KiB frames (for updating `struct page`), we are currently unable to fully harvest LLFREE's performance for 2 MiB frames. For example, while 4 KiB random frees are equally fast within the kernel, freeing a huge frame takes 4.48 times longer.

5.3 Multicore Scalability

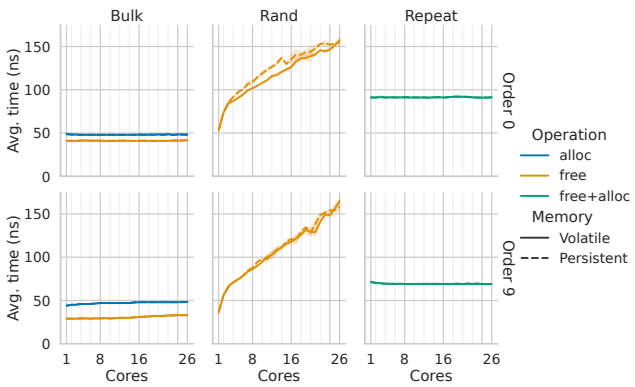


Figure 7: Average time per core count on orders 0 and 9 and 128 GiB memory of LLFREE in volatile and persistent memory

To evaluate multicore scalability, we focus on the two natural frame sizes and scale the number of requesting cores from 1 to 26. Again, Fig. 7 shows the raw LLFREE performance (in

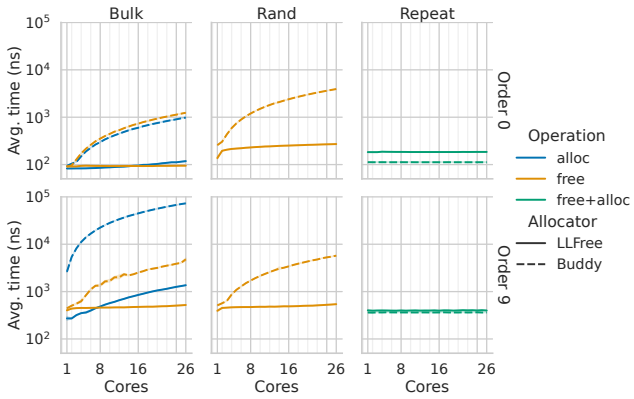


Figure 8: Average time per core count on orders 0 and 9 and 128 GiB memory in the Linux kernel on a logarithmic scale

userspace) on DRAM and NVRAM, while Fig. 8 (log-scale!) shows the in-kernel performance on DRAM.

For the bulk and repeat userspace benchmarks (Fig. 7), we see that LLFREE's operation times remain almost constant, independent of the number of cores, and the memory type has only an insignificant influence on the performance. Here, LLFREE's allocation and free reservation system avoids most sharing. Only for random, where cache invalidations and update conflicts on child counters are more frequent, we see a significant impact of more workers. However, even with 26 workers that request frames in parallel, a DRAM allocation/free of either order takes less than 170 ns.

In Fig. 8, we see that the Linux performance is heavily influenced by its per-CPU caches for the natural sizes. For the repeat benchmark, which is the best case for per-CPU caches, we see that LLFREE is up to 65.18 percent (26 cores, 4 KiB) slower than Linux. However, for bulk and random, which exceed the capacity of the per-CPU caches, the Linux allocator shows severe performance drops for more cores as memory is now requested directly from the buddy system. While the single-core performance for 4 KiB is still almost equal, Linux takes 7.3/13.5 (bulk allocations/random frees) *times* longer with 26 cores than LLFREE. For 26 cores requesting 2 MiB frames, this changes to 52.5/9.6 times.

5.4 List-Based Allocators

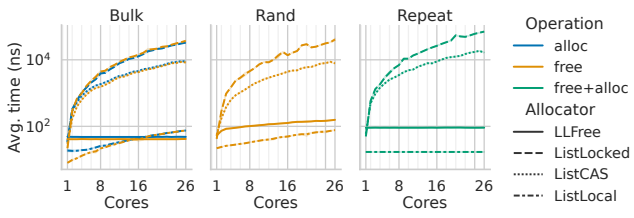


Figure 9: Average speed of the list and LLFREE allocators per core count for order 0 on 128 GiB DRAM

Besides the buddy system, simple free lists are a common design for page-frame allocators in commodity (Windows [50], Darwin [42]) and research (Twizzler [5]) operating systems. Like Linux, they mitigate lock contention on the global structures by additional core-local lists. To compare these allocator concepts with LLFREE, we built three prototypical list-based allocator implementations and evaluated their scalability. (1) The *ListLocked* allocator, consisting of a lock-based shared singly-linked list (Windows, Darwin), (2) the *ListCAS* allocator, which uses a LIFO lock-free list instead [44] (supposed to be preferable over locking), and (3) the *ListLocal* allocator, a theoretical allocator that maintains per-core lists only, does not need any protection and never drains (ideal case). All list allocators store their next pointers in a 64 B aligned array, similar to Linux's `struct page` array.

In Fig. 9, we see the results of this comparison on a logarithmic scale. As expected, the locked variant has the worst performance due to the high degree of lock contention. However, while replacing the lock with atomic operations improves the situation by 81 percent (26 cores, random), we see that this still does not solve the fundamental scaling issue; contention basically just moves from the lock to the cache line that contains the head pointer of the list.

To our surprise, LLFREE even outperforms (from 16 cores onwards) the ideal *ListLocal* variant, which is not even suited as a global frame allocator, but takes 36 percent more time on 26 cores for *bulk* allocations. This is caused by the state dispersion of linked lists: Virtually every allocation and free touches at least one new cache line – compared to LLFREE's cache-friendly structure, where in the best case, 512 allocations reuse the same three cache lines. In the *random* benchmarks, LLFREE also suffers from cache misses due to non-local frees (the worst case for LLFREE).

5.5 Allocator State Dispersion

To compare allocators' temporal and spatial costs, we propose the *state-dispersion* metric – a quantitative measure denoting the byte count utilized for metadata storage. Intuitively, state dispersion is the number of bytes accessed for a full enumeration of the internal state. However, it is critical to understand that state dispersion does not directly translate to memory overhead, given that allocators often repurpose the free memory or overload other shared data structures (i.e., `struct page`) for their metadata, whereby the plain memory overhead becomes a less meaningful metric. However, an allocator exhibiting high state dispersion will likely induce more cache misses during its operation, impacting run-time efficiency. This, of course, does not only depend on the size of the state but also its spatial distribution.

In Tab. 1, we break down the state dispersion of LLFREE and the Linux allocator into the different components. To put these numbers into perspective, we also show how the disper-

Allocator	Per Zone	Per CPU	Per 1 GiB	1 Zone 128 GiB 52CPUs	Full Scan: Accessed Cache Lines
LLFree				**4.1 MiB**	**67 754**
4 KiB Bit Fields			32.0 KiB	4.0 MiB	65 536
2 MiB Counters			1.0 KiB	128.0 KiB	2 048
64 MiB Trees		128 B	32 B	10.5 KiB	168
Global	128 B			128 B	2
Linux Buddy Allocator				**516.0 MiB**	**33 555 169**
Free Lists	988 B		4.0 MiB	512.0 MiB	33 554 448
Allocated Flag			32.0 KiB	4.0 MiB	(within above)
Pageblock Bits			256 B	32.0 KiB	512
Per-CPU Caches	8 B	256 B		13.0 KiB	209

Table 1: Allocator-State Dispersion and Cache Overhead

sion scales to our benchmark machine (52 cores, 128 GiB, 1 memory zone) and how many distinct 64B cache lines would be accessed for a complete enumeration in this setting.

Due to LLFree's usage of bit fields and counter arrays, its state disperses only to 4.14 MiB (0.0032 % of DRAM) on the benchmark machine. The primary contributor to this are the 4 KiB bit fields (4 MiB). Thus, even a full-state scan can comfortably fit within the machine's 35 MiB L3 cache, for which only 67 754 cache lines need to be loaded. Also, as LLFree does not repurpose memory, it does not require physical memory to be mapped by the kernel, and its state dispersion is equal to its memory overhead.

In contrast, the Linux allocator stores most of its state in `struct page`. There, it requires one flag and repurposes the 16 bytes of the LRU list pointers (double-linked list) for its per-order free lists and for the per-CPU page caches. Due to the scattered nature of linked lists, the Linux allocator (potentially) spreads its state over 516 MiB (0.39 % of DRAM). Even worse, as each `struct page` resides on its own cache line, a complete state scan would have loaded 33 555 169 cache lines. Hence, in comparison to LLFree, Linux's allocator state not only disperses over 125 times more memory, but even requires 495 times more cache lines to be loaded for the full scan.

Furthermore, as LLFree does not rely on the `struct page`, this also raises the question if they could be shrunk or eliminated. These records currently occupy 1.56 percent of DRAM. Unfortunately, removing the allocator's dependency on `struct page` does not directly result in smaller per-frame records, as other kernel subsystems reuse the LRU list pointers for various purposes when the frame is allocated (in the Linux source code, `struct page` is basically a mess of `unions`). Therefore, shrinking or even eliminating `struct page` is a deeply cross-cutting and challenging task.

Nevertheless, an allocator that, like LLFree, does not require a per-frame record significantly eases this challenge, since we then would only need per-frame records for *allocated* frames. For example, with Linux's current move to `struct folio` [11], which describes a bundle of physically contiguous frames, it could become possible to allocate this record dynamically. In this sense, we see LLFree as an important first step into untangling `struct page` dependencies. However, its complete elimination remains a topic of further research.

5.6 Fragmentation and Compaction Cost

Next, we look at the huge-frame fragmentation behavior of both allocators. For this, we first define a metric for this fragmentation and for the memory-compaction cost that would be required to remove this fragmentation. To measure fragmentation, we count the number of huge frames that can be allocated if we drain all caches but perform no memory compaction. We then compare this to the number of *possible* huge frames that could be allocated with compaction to get an idea of the fragmentation level in the system.

To gauge the compaction cost, we consider the minimal number of 4 KiB copy operations required to free up the *possible* maximum of huge frames. To calculate this metric, we (1) count the number of free 4 KiB frames in each possible huge frame, (2) sort the resulting array, and (3) match the "fullest" with the "emptiest" huge frames while counting the number of required copy operations. Please note that Linux skips step 1 and 2 and moves memory to the beginning of the zone, which results in suboptimal memory compaction. With LLFree, however, sorting the children array (see Fig. 3) would bring us close to the optimal variant.

To compare both allocators, we conduct the following synthetic benchmark: First, we generate an initial memory configuration that is a worst-case scenario for a maximally fragmented physical memory. For this, we allocate 90 percent of a 125 GiB region before freeing up half of all 4 KiB frames randomly again. Starting with this fragmented state, we perform 100 iterations, each freeing 10 percent of the allocated memory in the form of randomly-selected 4 KiB frames and reallocating the same amount again as individual 4 KiB frames. After each iteration, we drain the CPU-local caches (buddy), respectively the tree reservations (LLFree), and measure the huge-frame fragmentation and the compaction cost. Note that we still leave the Linux memory compactor turned off (as stated in Sec. 4.2.2). Like in our other benchmarks, we do not trigger unfulfillable allocation requests, which would require synchronous compaction. We measured the hypothetical compaction costs for each iteration for both allocators using the described offline calculation.

Fig. 10 shows the change of both metrics over time. The Linux allocator can recover only a single huge frame in this benchmark, although the whole memory was cycled ten times over (100 iterations). Also, the compaction cost decreased only by 3.3 percent, indicating that huge frames are only getting slowly defragmented over time. Additionally, our scenario benefits Linux as we drain the per-CPU caches and use the optimal compaction-cost metric. In contrast, LLFree can recover 46.6 percent of the initially polluted huge frames over

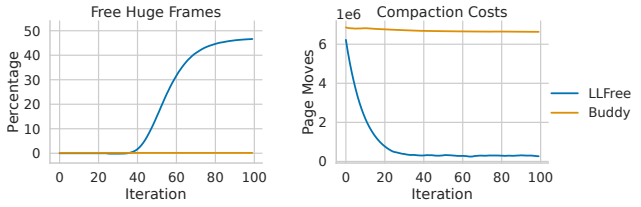


Figure 10: Free huge frames (left) and compaction cost (right) over iterations that randomly reallocate 10% of the allocated memory.

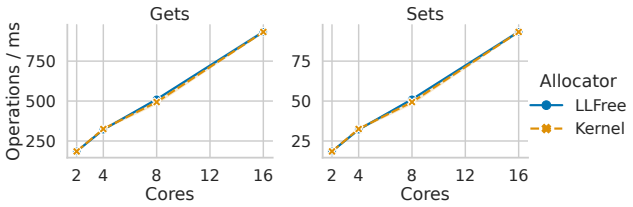


Figure 11: Number of Gets / Sets per millisecond for the *memtier* benchmark.

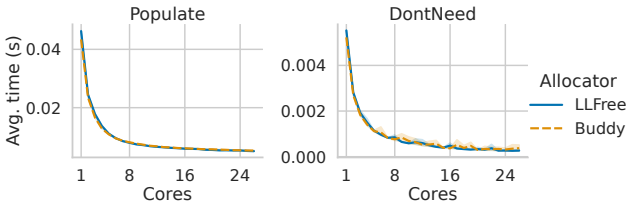


Figure 12: Average time to populate/free a 128 GiB memory map in the *write* benchmark.

the benchmark. Although it looks like defragmentation only starts to kick in around iteration 50, a look at the compaction cost indicates that entropy decreases right from the beginning; there are just no completely free huge-frame frames yet. After performing reallocations summing up to the total amount of memory (10 iterations), we are at 39.1 percent compaction cost and after 50 iterations, we would only require 4.9 percent of the initially-required copy operations.

Overall, we see that LLFREE shows a passive defragmentation behavior steered by our subtree-allocation policy. As a buddy system does not track the "fullness" of split-up buckets, it cannot imitate this on the cheap.

5.7 Crash Recovery

As validating crash consistency by real system crashes is too time-consuming to obtain robust results, we simulate LLFREE's recovery (Sec. 3.2.3) for regular shutdowns and crashes using a userspace benchmark on a DAX-mapped NVRAM region: For this, (1) we initialize the allocator on a 128 GiB region, (2) allocate half of it randomly, and (3) allocate and free memory repeatedly as in Sec. 5.6. For regular shutdowns, the process terminates after (2), while we simulate crashes by randomly killing the benchmark with `SIGKILL` during (3). Afterward, another process recovers the allocator's state from the persistent memory.

In total, we injected 1 000 crashes and LLFREE could recover its state in all cases; in about half of the experiments, we actually lost frames (at most one per core), which is expected, as the cores spent all their time in alloc/free. For the recovery procedure, LLFREE iterates through the bit fields to correct the child counters, which takes on average 2 460 µs. In comparison, a regular NVRAM re-initialization, where LLFREE only needs to iterate over the child tables, takes only 477 µs. Recovery was done single-threaded but could be parallelized and partially performed in the background if recovery times should become an issue.

5.8 Application-Level Benchmarks

As a real-world application benchmark, we used `memtier`[6], which evaluates the performance of the *memcached* key-value store. It measures the throughput of *Get* and *Set* requests. Unfortunately, we see no significant difference as shown in Fig. 11. As the page allocator is primarily used by the page-fault handler, which lazily allocates memory, the overhead of the other involved memory management components might overshadow any performance gains.

To investigate this hypothesis further, we created the *write* benchmark, which maps a large memory region and populates it in parallel. The population is done by writing a non-zero value into the first byte of the page, triggering a page fault and subsequent allocation request. For unmapping, the `madvice/DONTNEED` syscall is used. The benchmark is executed for 1–26 cores, with the memory region split evenly between the cores. Again, the results in Fig. 12 show no significant difference between the buddy and LLFREE allocators, just like the *memtier* benchmark.

Utilizing the *perf* profiler, we measured where most of the runtime is spent. The flame graph in Fig. 13 shows that the page allocator (yellow) only accounts for 5.3 percent of the runtime. Primarily dominant are the `struct mm` rw-lock (orange, 17.3 %) and the updates of the LRU, cgroups, reverse mappings, and `struct page` flags (green, 28.7 %). Because most `struct page` update macros are inlined, the actual time is probably higher. These memory-management bottlenecks are consistent with other research [7, 12, 14, 32, 35].

6 Discussion

Our results show (besides crash consistency) that LLFREE provides excellent scalability in user- and kernel-level benchmarks, achieved by its consequent lock-free and cache-friendly design. Nevertheless, these benefits are not yet visible in end-to-end benchmarks, even though there are applications

[6] https://github.com/RedisLabs/memtier_benchmark

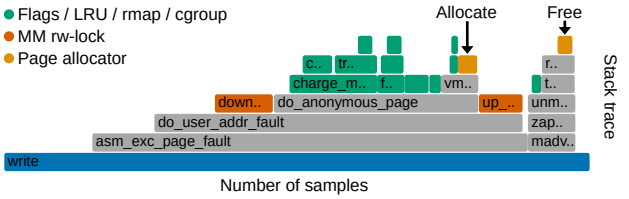


Figure 13: Flame graph for the write benchmark with the Buddy allocator on 8 cores and 128 GiB DRAM.

for which OS-level memory de/allocation performance already *is* a big issue [12, 14, 32, 35].

We argue that, given the deep entangling and grown complexity, the scalability problem could only be solved in a bottom-up manner and provide LLFREE and its design concepts as a first step in this direction. We believe if nonvolatile memories will play a role *some* day, their kernel-level management has to be designed together with volatile memory – and see LLFREE as an important step in this direction. Finally, our results also show that the conjunction of striving for scalability *and* persistence [8, 16, 46] works out particularly well in kernel design if considered from the very beginning.

7 Related Work

Many general-purpose allocators [3, 9, 33, 36, 41, 45, 52] for nonvolatile memory have been proposed. In contrast to LLFREE, all of them use logging to ensure crash consistency, which increases NVM wear [49], and locks for multithreaded operation; some reduce lock contention by using multiple allocator instances [36, 41], per-CPU/thread free lists [3, 9] or range locking [52]. From these, *PAllocator* [36] has similarities to LLFREE as it comes with antifragmentation measures and, similar to our lower level, stores only parts of its state in NVRAM and recovers its volatile state on boot. Nevertheless, all of these persistent allocators are general-purpose userspace allocators and, thus, have different design goals compared to a kernel page-frame allocator like LLFREE.

On the OS side, *Twizzler* [5] is explicitly built around nonvolatile memory but nevertheless does not contain a persistent page-frame allocator. Instead, the system rebuilds the allocator state from the persistent objects on each reboot in DRAM. To our knowledge, LLFREE is the first persistent page-frame allocator to be used within the operating system.

While the immunity to external fragmentation was one of the original motivations for paging [2], its extension to different frame sizes brought back the problem. To ease active huge-frame reclamation, placement strategies [19–21] categorize allocations (i.e., movable, reclaimable) and spatially cluster them onto separate huge frames. For this, Linux has multiple free lists per buddy order, each of which serves a different category. LLFREE currently does not support such categorization. However, it could easily be extended for this by specialized trees (see Sec. 3.2.2). For Linux, strategies with better clustering characteristics have been suggested [37].

Other measures to reduce huge-frame fragmentation include proactive compaction [30] and anticipated continuous memory reservation [27, 34]. Even hardware solutions have been proposed, such as building huge frames of noncontinuous memory [43], or an additional level in address translation [51] similar to nested paging [4]. In contrast, LLFREE is a pure software solution that passively defragments huge frames while being fast and crash-consistent at the same time.

8 Conclusion

The page-frame allocator, which manages the physical memory, is at the core of all memory management in modern operating systems. However, as we have shown in the example of Linux, its classical lock-based design with many lists and distributed metadata has not kept up with the progress in hardware towards massive-parallel systems with large amounts of heterogeneous volatile and nonvolatile memories. This results in internal complexity, poor scalability, high memory fragmentation, and general unfitness for achieving crash-consistent allocations on nonvolatile memories.

We presented LLFREE, a new log- and lock-free page-frame allocator that, by its lockless and cache-centric design, achieves excellent scalability for parallel allocations (53 times faster than the Linux buddy allocator for parallel 2 MiB DRAM allocations on 26 cores), while constructively keeping huge-page fragmentation low. All de/allocations manifest in memory by a one–cache-line transaction, whereby LLFREE can provide crash-consistency for persistent NVRAM without logging and at near-DRAM speed on eADR systems. Our integration of LLFREE into Linux was successful, but it also revealed many further bottlenecks of its memory-management subsystem and the deep entangling of the buddy allocator with it. These topics demand further investigation and redesign. We consider LLFREE a crucial first step towards a complete structural rethinking of the OS memory management.

Acknowledgments

We thank our anonymous reviewers for their helpful and constructive comments. Special thanks go to Godmar Back, whose demanding and encouraging shepherding has helped us tremendously improve this paper's content and quality.

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 468988364, 501887536.

References

[1] AKRAM, S. Exploiting Intel Optane persistent memory for full text search. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2021), ISMM 2021, Association for Computing Machinery, p. 80–93.

[2] BENSOUSSAN, A., CLINGEN, C. T., AND DALEY, R. C. The Multics virtual memory. In *Proceedings of the 2nd ACM Symposium on Operating Systems Principles (SOSP '69)* (New York, NY, USA, 1969), ACM Press, pp. 30–42.

[3] BHANDARI, K., CHAKRABARTI, D. R., AND BOEHM, H. Makalu: fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016* (2016), pp. 677–694.

[4] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, Association for Computing Machinery, p. 26–35.

[5] BITTMAN, D., ALVARO, P., MEHRA, P., LONG, D. D. E., AND MILLER, E. L. Twizzler: a data-centric OS for non-volatile memory. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)* (July 2020), USENIX Association, pp. 65–80.

[6] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *8th Symposium on Operating System Design and Implementation (OSDI '08)* (Berkeley, CA, USA, 2008), USENIX Association, pp. 43–57.

[7] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *9th Symposium on Operating System Design and Implementation (OSDI '10)* (Berkeley, CA, USA, 2010).

[8] CHEN, Z., HUA, Y., DING, B., AND ZUO, P. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 799–812.

[9] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News 39*, 1 (Mar. 2011), 105–118.

[10] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)* (New York, NY, USA, 2009), ACM, pp. 133–146.

[11] CORBET, J. Clarifying memory management with page folios, 05 2023.

[12] CROTTY, A., LEIS, V., AND PAVLO, A. Are you sure you want to use mmap in your database management system? In *CIDR 2022, Conference on Innovative Data Systems Research* (2022).

[13] DAVID, T., DRAGOJEVIĆ, A., GUERRAOUI, R., AND ZABLOTCHI, I. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 373–386.

[14] DURNER, D., LEIS, V., AND NEUMANN, T. Experimental study of memory allocation for high-performance query processing. In *International Conference on Very Large Databases (VLDB)* (2019), pp. 1–9.

[15] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (New York, NY, USA, Dec. 1995), ACM Press, pp. 251–266.

[16] FRIEDMAN, M., HERLIHY, M., MARATHE, V., AND PETRANK, E. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2018), PPoPP '18, Association for Computing Machinery, pp. 28–40.

[17] FU, X., KIM, W.-H., SHREEPATHI, A. P., ISMAIL, M., WADKAR, S., LEE, D., AND MIN, C. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 100–115.

[18] GORMAN, M. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[19] GORMAN, M. The performance and behaviour of the anti-fragmentation related patches. Linux Kernel Mailing List, Mar. 2007. https://lkml.org/lkml/2007/3/1/92.

[20] GORMAN, M., AND HEALY, P. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management - ISMM '08* (Tucson, AZ, USA, 2008), ACM Press, p. 41.

[21] GORMAN, M., AND WHITCROFT, A. The what, the why and the where to of anti-fragmentation. In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, Jul 2006), vol. Volume 1, p. 370–384.

[22] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., SCHÖNBERG, S., AND WOLTER, J. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (New York, NY, USA, Oct. 1997), ACM Press.

[23] HAYOT-SASSON, V., BROWN, S. T., AND GLATARD, T. Performance benefits of intel optane dc persistent memory for the parallel processing of large neuroimaging data. *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)* (2019), 509–518.

[24] INTEL. eADR: new opportunities for persistent memory applications, 2021. https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html, visited 2021-10-04.

[25] INTEL. 2022 Q2 – 10-Q earnings report, 2022. https://www.intc.com/filings-reports/all-sec-filings/content/0000050863-22-000030/0000050863-22-000030.pdf.

[26] IZRAELEVITZ, J., MENDES, H., AND SCOTT, M. L. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing* (2016), Springer, pp. 313–327.

[27] KIM, S.-H., KWON, S., KIM, J.-S., AND JEONG, J. Controlling physical memory fragmentation in mobile systems. In *Proceedings of the 2015 International Symposium on Memory Management* (New York, NY, USA, 2015), ISMM '15, Association for Computing Machinery, p. 1–14.

[28] KNOWLTON, K. C. A fast storage allocator. *Communications of the ACM 8*, 10 (1965), 623–624.

[29] KORGAONKAR, K., IZRAELEVITZ, J., ZHAO, J., AND SWANSON, S. Vorpal: Vector clock ordering for large persistent memory systems. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 435–444.

[30] KWON, Y., YU, H., PETER, S., ROSSBACH, C. J., AND WITCHEL, E. Coordinated and efficient huge page management with ingens. In *12th Symposium on Operating Systems Design and Implementation (OSDI '16)* (USA, 2016), USENIX Association, p. 705–721.

[31] LEE, S., KWON, M., PARK, G., AND JUNG, M. Lightpc: Hardware and software co-design for energy-efficient full system persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2022), ISCA '22, Association for Computing Machinery, p. 289–305.

[32] LEIS, V., ALHOMSSI, A., ZIEGLER, T., LOECK, Y., AND DIETRICH, C. Virtual-memory assisted buffer management. In *Proceedings of the ACM SIGMOD-/PODS International Conference on Management of Data (SIGMOD'23)* (New York, NY, USA, June 2023), ACM.

[33] MEMARIPOUR, A. S., IZRAELEVITZ, J., AND SWANSON, S. Pronto: Easy and fast persistence for volatile data structures. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), pp. 789–806.

[34] NAVARRO, J., IYER, S., AND COX, A. Practical, transparent operating system support for superpages. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)* (Boston, MA, Dec. 2002), USENIX Association.

[35] NEUMANN, T., AND FREITAG, M. J. Umbra: A disk-based system with in-memory performance. In *Conference on Innovative Data Systems Research (CIDR)* (2020).

[36] OUKID, I., BOOSS, D., LESPINASSE, A., LEHNER, W., WILLHALM, T., AND GOMES, G. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow. 10*, 11 (aug 2017), 1166–1177.

[37] PANWAR, A., PRASAD, A., AND GOPINATH, K. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, Association for Computing Machinery, p. 679–692.

[38] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)* (2014), IEEE Press, p. 265–276.

[39] PMDK TEAM, I. C. Intel Persistent Memory Develpment Kit (PMDK). https://pmem.io/pmdk, visited 2021-10-05.

[40] RAO, D. S., KUMAR, S., KESHAVAMURTHY, A. S., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth ACM European Conference on Computer Systems (EuroSys '14)* (2014), pp. 15:1–15:15.

[41] SCHWALB, D., BERNING, T., FAUST, M., DRESELER, M., AND PLATTNER, H. nvm malloc: Memory allocation for NVRAM. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015* (2015), R. Bordawekar, T. Lahiri, B. Gedik, and C. A. Lang, Eds., pp. 61–72.

[42] SINGH, A. *Mac OS X Internals: A Systems Approach: A Systems Approach*. Addison Wesley, 2016.

[43] SWANSON, M., STOLLER, L., AND CARTER, J. Increasing TLB reach using superpages backed by shadow memory. *SIGARCH Comput. Archit. News 26*, 3 (apr 1998), 204–213.

[44] TREIBER, R. K. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research, 1986.

[45] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: lightweight persistent memory. *ACM SIGARCH Computer Architecture News 39*, 1 (Mar. 2011), 91–104.

[46] WANG, T., LEVANDOSKI, J., AND LARSON, P.-A. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 461–472.

[47] WENTZLAFF, D., AND AGARWAL, A. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review 43* (Apr. 2009), 76–85.

[48] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST '16)* (Santa Clara, CA, 2016), USENIX Association, pp. 323–338.

[49] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[50] YOSIFOVICH, P., RUSSINOVICH, M., SOLOMON, D. A., AND IONESCU, A. *Windows Internals, Part 1 (7th Edition)*. Microsoft Press, 2017.

[51] ZHANG, L., SPEIGHT, E., RAJAMONY, R., AND LIN, J. Enigma: Architectural and operating system support for reducing the impact of address translation. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)* (New York, NY, USA, 2010), ICS '10, Association for Computing Machinery, p. 159–168.

[52] ZHANG, L., AND SWANSON, S. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 897–912.

Appendix

(1) If reserved counter $c_P \geq 2^o$ then $c_P \leftarrow c_P - 2^o$ and continue with (2).
  (a) Otherwise, sync with global c_U and repeat (1) if the counter is now large enough.
  (b) Otherwise, reserve a new tree and repeat (1).
(2) Search the corresponding children array sequentially for an entry with $c_L \geq 2^o$. If this search fails, reserve a new subtree and repeat (1).
  (a) For base frames, decrement c_L, search the corresponding bit field for a zero bit, and set it.
  (b) For huge frames set $c_L = 0$ and $a = 1$.
(3) Return the allocated *page-frame number (PFN)* or `NULL`.

(a) The allocation of an order o frame.

(1) Check the corresponding child entry.
  (a) For base frames, check if $c_L \leq 512 - 2^o$ and $a = 0$, and continue with (2).
  (b) For huge frames, check for $a = 1$, set it to zero, and continue with (4).
(2) Toggle the corresponding bits in the layer-one bit field.
(3) Increment the child counter ($c_L \leftarrow c_L + 2^o$).
(4) Increment the reserved c_P or the global c_U if this free is in another tree.
  (a) When a global, *partial* tree entry is updated, reserve it if the past F allocations also affected it.

(b) The free operation of an order o frame.

Figure 14: The allocation and free algorithms. This description does *not* include all edge cases and error paths.

A Artifact Appendix

Abstract

The artifact contains the necessary tools and resources required to evaluate LLFree, a new lock- and log-free allocator design that scales well, has a small memory footprint, and is readily applicable to non-volatile memory. To simplify the evaluation, the artifact is packaged as a Docker image, which includes the different benchmarks from the paper's evaluation and any dependencies. These benchmarks are designed to stress the allocator in various scenarios. They allow other researchers to compare the performance of LLFree with the traditional Buddy allocator and reproduce our experimental results. Additionally, this image contains the raw data and scripts for the paper's figures, making our evaluation as transparent as possible.

Scope

These benchmarks show that the LLFree allocator out scales the buddy allocator on systems and workloads with high parallelism. However, executing them in a virtual machine (even with KVM) leads to less accurate results. Therefore, in the paper, we built and tested the modified Linux on raw hardware. Nonetheless, the results should show similar trends as in the paper's evaluation.

Contents

The Docker image includes all required dependencies and scripts for building and running the benchmarks, as well as generating relevant plots and data. It also features a Python script (run.py) that serves as the central command center to manage the building, benchmarking, and plotting processes. The allocator can be tested in both user space and within a custom-built Linux kernel that incorporates LLFree executed in a QEMU+KVM vm. For the latter, the image contains a QEMU+KVM virtual machine and scripts to boot it and run the kernel benchmarks. It further contains the raw data and plots from the benchmarks shown in the paper.

Hosting

The docker image and the repositories of the allocator, the modified Linux kernel, and the benchmarks are hosted on GitHub:

- **Docker Image**: This image contains an execution environment that makes it easy to run and evaluate the benchmarks.
- **llfree-bench**: The benchmark scripts and results.
  (tag = `atc23-artifact-eval`)
  - The artifact instructions can be found in `artifact-eval/README.md`.

- **llfree-rs**: The Rust implementation of the LLFree allocator.
  (tag = `atc23-artifact-eval`)
- **llfree-linux**: The modified Linux Kernel that can be configured to use LLFree instead of the Buddy allocator.
  (tag = `atc23-artifact-eval`)
- **linux-alloc-bench**: Kernel module for benchmarking the page allocator.
  (tag = `atc23-artifact-eval`)

Requirements

As our benchmarks are packaged in a Docker image and do not rely on specific hardware, the only prerequisites are:

- A Linux-based system for KVM. We have tested this on Linux 6.0, 6.1, and 6.2.
- At least 8 physical cores and 32GB RAM (more is better). Lower specifications should work, but the results may be less meaningful.
- Hyperthreading and TurboBoost should be disabled for more stable results. As the VM is not configured for this, the kernel benchmarks might be especially affected.
- A properly installed and running Docker daemon.

Next Steps

The artifact instructions can be found in the *llfree-bench* repository under artifact-eval/README.md. The *artifact-eval* directory also contains all the scripts mentioned in the document and also the Dockerfile for building the image by oneself if desired.

SingularFS: A Billion-Scale Distributed File System Using a Single Metadata Server

Hao Guo Youyou Lu* Wenhao Lv Xiaojian Liao Shaoxun Zeng Jiwu Shu
Tsinghua University

Abstract

Billion-scale distributed file systems play an important role in modern datacenters, and it is desirable and possible to support these file systems with a single metadata server. However, fully exploiting its performance faces unique challenges, including crash consistency overhead, lock contention in a shared directory, and NUMA locality.

This paper presents SingularFS, a billion-scale distributed file system using a single metadata server. It includes three key techniques. First, SingularFS proposes *log-free metadata operations* to eliminate additional crash consistency overheads for most metadata operations. Second, SingularFS designs *hierarchical concurrency control* to maximize the parallelism of metadata operations. Third, SingularFS introduces *hybrid inode partition* to reduce inter-NUMA access and intra-NUMA lock contention. Our extensive evaluation shows that SingularFS consistently provides high performance for metadata operations on both private and shared directories, and has a steadily high throughput for the billion-scale directory tree.

1 Introduction

In modern datacenters, the vast majority of distributed file systems are within billions of files, and we call them billion-scale distributed file systems [33]. It is possible to support these file systems with one single metadata server, which typically has enough capacity to hold terabytes of metadata. However, the performance of a single metadata server requires further attention, as metadata operations account for more than half of all file system operations [16, 17, 26].

We find it challenging to store billions of files without sacrificing performance in a single metadata server, so most distributed file systems support billions of files by scaling metadata servers [17, 19, 22, 25, 31]. In this paper, we explore the design space of a billion-scale distributed file system that achieves high performance in a single metadata server.

Remote direct memory access (RDMA) and persistent memory (PM) provide new opportunities for the performance of metadata servers. However, existing solutions fail to fully exploit them and provide desirable metadata performance. Our experiments show that both local PM file systems and distributed file systems have performance limitations in both private and shared directories. Specifically, for the state-of-the-art local PM file system, NOVA [32], its file `create` throughput in a shared directory drops to only 0.14× of its throughput in private directories, even for the million-scale directory tree without the impact of networking.

Exploiting the performance of a single metadata server brings several challenges to the design of file systems. First, the overhead to ensure crash consistency is extremely heavy for a single metadata server that aims to support billions of files. Second, operations in a shared directory, which are common in distributed file systems, suffer from limited parallelism and low performance caused by severe lock contention on the directory `dirent`s and inode. Third, The NUMA architecture is under-exploited for file systems, especially for metadata operations in a single metadata server.

This paper presents SingularFS, a billion-scale distributed file system using a single metadata server. With only one metadata server, SingularFS achieves 8.36M/18.80M IOPS for file `create`/`stat` operations, which outperforms InfiniFS with 32 metadata servers reported in its paper [19], without sacrificing multi-server scalability. To address the challenges mentioned above, SingularFS optimizes the directory tree and the metadata operations with the following designs.

First, we propose *log-free metadata operations* to remove the additional crash consistency overheads from most metadata operations. The key idea is to identify possible metadata inconsistency by leveraging both the single-object update atomicity of the key-value storage backend and the metadata semantic dependency of the parent inode and child inodes.

Second, we design *hierarchical concurrency control* to maximize the parallelism of metadata operations in a shared directory with less synchronization overhead. The key idea of this protocol is to synchronize inode operations in a more fine-grained way. Specifically, inode *writer* uses the per-inode read-write lock to synchronize with other operations, and

*Youyou Lu is the corresponding author (luyouyou@tsinghua.edu.cn).

inode timestamp *updater* and *reader* use a lock-free protocol to do extra synchronization between themselves.

Third, we introduce *hybrid inode partition* to reduce inter-NUMA access and intra-NUMA lock contention. The key idea is to separate the timestamps from the directory inode and group it with the directory's child inodes to the same NUMA node, thus ensuring NUMA locality of file operations. SingularFS further partitions the intra-NUMA data structure to reduce its lock contention.

In summary, this paper makes the following contributions:

- We identify the challenges to fully exploiting the performance of a single metadata server.
- We propose SingularFS, an efficient distributed file system using a single metadata server, featured with log-free metadata operations (§3.2), hierarchical concurrency control (§3.3), and hybrid inode partition (§3.4).
- We implement and evaluate SingularFS to demonstrate that SingularFS outperforms existing distributed file systems in latency and throughput of metadata operations, has comparable latency and throughput with local PM file systems in file operations, provides high throughput scalability in a shared directory, and maintains a steadily high throughput for a billion-scale directory tree (§5).

2 Background and Motivation

2.1 Background

Billion-scale file systems are fundamental building blocks for cloud service vendors and smaller datacenters. Even in some huge datacenters such as Alibaba [19], massive files are managed with small storage clusters, which are within billion-scale. One single metadata server is sufficient to contain all the metadata at this scale, and it has the following benefits compared to using more metadata servers:

- *Implementation and performance.* Distributed transactions and load balancing across metadata servers are avoided, which simplifies the implementation and improves the performance.
- *TCO reduction.* The installation, maintenance, and daily cost of a single metadata server are cheaper than multiple metadata servers.

New network and storage hardware, such as RDMA and PM, provides new opportunities for metadata performance. NVIDIA's latest generation NIC, ConnectX-6, exhibits a speed of 215Mpps for small packets [4]. RedN [23] also illustrates that ConnectX-6 shows a 112M verbs/s throughput in 64B RDMA writes. The only available PM product, Intel Optane DIMM, shows at least 29.06Mops/s and 8.75Mops/s read/write throughput with the access granularity of less than 256B [2] while maintaining memory semantics and data persistence. As inodes are typically tiny in file systems (e.g., 128B in Ext4), achieving high small-granularity access performance allows us to build a high-performance distributed file system with a single metadata server.

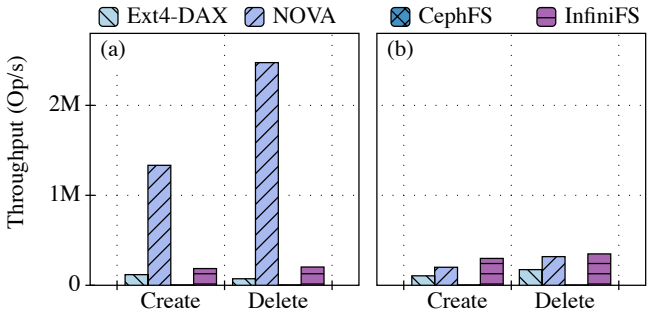


Figure 1: Per-NUMA file `create` and `delete` throughput of local PM file systems and distributed file systems (a) in a private directory for each client, (b) in a shared directory.

2.2 Analysis of Existing Solutions

In this section, we analyze the limitations of existing solutions that discourage them to support a billion-scale distributed file system with a single metadata server.

There are mainly two categories of existing solutions, local PM file systems, and distributed file systems. For each type, we select two typical file systems for comparison, namely Ext4-DAX and NOVA [32] for local PM file systems, and CephFS [31] and InfiniFS [19] for distributed file systems. We gradually increase the number of client threads until each file system achieves the peak throughput. The testbed and configuration details are further described in §5.1.

Figure 1 shows the per-NUMA file `create` and `delete` throughput of the compared file systems with clients operating on private directories and a shared directory, respectively. We make the following observations:

1) The overhead of crash consistency limits the throughput of multi-inode metadata update operations. Ext4-DAX uses write-ahead-logging (WAL) to guarantee its crash consistency, while InfiniFS and CephFS leverage the transaction support of their storage backend. These methods all introduce extra costs for the multi-inode metadata update operations. NOVA reduces the crash consistency cost by using the log-structured metadata architecture, making its file `create`/`delete` throughput 7.17×/12.19× higher than other file systems with private directories. However, it cannot completely get rid of the journaling overhead when coordinating multi-inode update operations. Also, it introduces extra garbage collection overhead.

2) The lock contention limits the throughput of metadata operations in a shared directory. For file `create` and `delete` operations, NOVA shows 0.17×/0.13× throughput in a shared directory than in private directories. This is because NOVA directly acquires the shared directory's write lock when performing these operations, which aggravates lock contention. The other three file systems show no significant performance

degradation. This is because they either directly use the journal for concurrency control (e.g., Ext4-DAX) or leverage the transaction support of the storage backend (e.g., CephFS and InfiniFS). These methods coordinate all the metadata operations similarly no matter they are in private directories or a shared directory, limiting the performance in both scenarios.

3) File systems fail to scale to multiple NUMA nodes while maintaining NUMA locality of metadata operations. For the compared file systems, NOVA doesn't provide support for multiple NUMA nodes. Although Ext4-DAX can be mounted to a RAID 0 device spanning across the PM DIMMs on all NUMA nodes, the simple striped layout of RAID 0 can not ensure NUMA locality for metadata operations. For InfiniFS and CephFS, they scatter the metadata objects to all the PM DIMMs without specific NUMA-aware partition rules, resulting in low NUMA locality as well.

2.3 Challenges

Based on the three observations in §2.2, we find three challenges to fully exploiting the performance of a single metadata server in distributed file systems, as discussed below.

Challenge 1. *The overhead to ensure crash consistency is extremely heavy for a single metadata server that aims to support billions of files.*

File systems use journaling or the log-structured design to provide crash consistency. In the journaling approach, data is written twice and checkpointed in order. In the log-structured approach, data is written in newly-allocated places, while leaving the old places as garbage. Unfortunately, garbage collection causes high overhead. Intensive prior research aims to reduce the overhead of the crash consistency mechanisms [14, 18, 21]. However, we need to further reduce this overhead to exploit the potential of a single server to support billions of files.

Challenge 2. *Operations in a shared directory, which are common in distributed file systems, suffer from limited parallelism and low performance caused by severe lock contention on the directory* `dirent`*s and inode.*

In HPC and big data applications, it is common for the workloads to concurrently access a large shared directory, such as N-N checkpointing [9] and image processing [1]. The concurrency of metadata operations in a shared directory significantly influences the overall performance of these applications. However, concurrent inode `create` and `delete` operations in a shared directory need to update their common parent's directory entries (`dirent`s) and inode, which causes high lock contention. Such contention limits the parallelism and performance as demonstrated in §2.2.

The previously-proposed solutions are not suitable for the case of a single metadata server. Previous works use partition strategy either inside `dirent`s [34] or between metadata server daemons [22, 31] to increase parallelism. However, partitioning inside `dirent`s can't reduce the lock contention of the common parent inode, and partitioning the super directories between metadata server daemons can not be used for one metadata server.

Challenge 3. *The NUMA architecture is under-exploited for file systems, especially for metadata operations in a single metadata server.*

NUMA-aware design is important for metadata performance when a server is used only for metadata storage and processing. On the one hand, NUMA locality is of great importance for fully utilizing the performance potentials of PM. Previous works illustrate that remote PM access introduces a significant performance drop in bandwidth and small-granularity throughput, especially for writes [28, 35], which makes metadata operations more vulnerable compared to data operations. On the other hand, our analysis in §2.2 shows that file systems fail to scale to multiple NUMA nodes while keeping NUMA locality of metadata operations, which is because of their coarse-grained partition strategy such as striping.

Existing methods are not desirable for overcoming this challenge. Randomly scattering the inodes into multiple NUMA nodes can't avoid this issue, as some operations like inode `create` and `delete` need to update multiple inodes, which causes inter-NUMA metadata access. Other methods like in-DRAM cache [28], thread migration [30], or thread delegation [35] all waste extra resources for coordination or data structure maintenance.

3 Design and Implementation

With the overall goal of exploiting the performance of a single metadata server, we design SingularFS with three key design principles as below:

- **Guaranteeing crash consistency without logs.** SingularFS performs most metadata operations without extra crash consistency mechanisms such as logs. This reduces their overheads while still maintaining POSIX semantics.
- **Maximizing parallelism in a shared directory.** By utilizing hierarchical concurrency control, SingularFS maximizes the parallelism of metadata operations in a shared directory with less synchronization overhead.
- **Reducing inter-NUMA access and intra-NUMA lock contention.** SingularFS takes a hybrid approach to metadata partition. This ensures NUMA locality of file operations and reduces lock contention within data structures to further increase the parallelism in a shared directory.

3.1 Overview

SingularFS is a billion-scale distributed file system using a single metadata server, exploiting the single-server performance while not sacrificing multi-server scalability. Figure 2

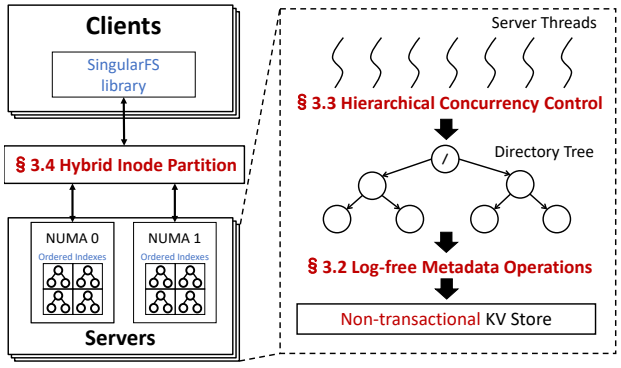


Figure 2: SingularFS architecture.

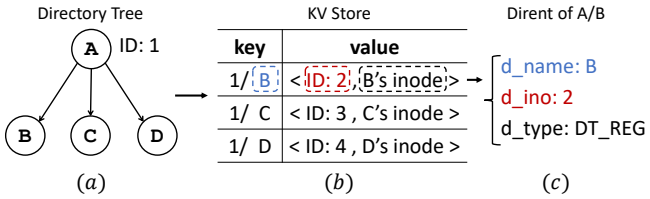


Figure 3: (a) An example of the directory tree. (b) The format of directory A's child inodes in KV Store. For the inode with ID = *i*, the keys of its child inodes share the same prefix *i*. (c) The way to reassemble `dirents` in a `readdir` operation. SingularFS first does prefix matching with the target directory's ID to get the child KV pairs, then uses the keys to generate `d_name` and reads the values to get `d_ino` and `d_type`.

presents the architecture of SingularFS, which contains two components, clients and servers. Servers maintain a global file system directory tree inside PM. Clients perform file system operations through POSIX-like interfaces offered by the user-space library. Servers and clients are equipped with RDMA NICs for network communication.

Storage backend. SingularFS uses a generic key-value store (KV Store) as its storage backend. The KV Store should be able to execute point queries and prefix matching. Besides, it should guarantee the atomicity of single-object operations at a low runtime cost.

SingularFS differs from other file systems with KV Store backends [17, 19, 24, 25] in two ways. First, instead of relying on the transaction provided by the KV Store backends (Non-transactional KV Store in Figure 2), SingularFS uses a lightweight approach for metadata operations (§3.2, §3.3). Second, instead of using one ordered KV Store per metadata server, SingularFS leverages hybrid inode partition to convert the shared index into multiple ordered indexes scattered to NUMA nodes, aiming at reducing inter-NUMA access and intra-NUMA lock contention (§3.4).

Regular metadata operations. We use this term to refer to metadata operations other than `rename`. Leveraging the timestamp dependency of modified inodes, SingularFS uses ordered metadata update to guarantee their crash consistency without logs (§3.2). For concurrency control, SingularFS takes a hierarchical method by utilizing both the per-inode read-write lock and lock-free timestamp update (§3.3).

Rename. SingularFS uses journaling to guarantee the crash consistency of `rename`. As an optimization, ordered metadata update (§3.2) is adopted to guarantee the crash consistency of the two parent directories involved in `rename`, reducing the number of logged inodes from 4 to 2. SingularFS uses strict two-phase locking for concurrency control of `rename`.

Inode. SingularFS adds two fields, the born time *btime* and the death time *dtime*, to the inode to identify metadata inconsistencies leveraging the timestamp dependency (§3.2). Besides, SingularFS partitions the directory inode to timestamp metadata (`atime`, `ctime`, and `mtime`) and access metadata (inode ID, permission, *btime*, *dtime*, etc.), and places the timestamp metadata of the parent directory with its child inodes into the same NUMA node to ensure NUMA locality of file operations (§3.4).

Inodes are all stored in the KV Store backend. Directory access metadata and file inodes are indexed by key <`parent inode ID`+`name`>, and directory timestamp metadata is indexed by key <`inode ID`>. The root directory has a unique inode ID 0. Path resolution is done by recursively fetching the directory access metadata from the root to the target inode.

Dirent. Directory entries (`dirents`) are omitted from directory metadata blocks but co-located with the keys and values of child inode objects. Thus, `dirent` update and inode update are fused into one key-value update operation. As shown in Figure 3, in a `readdir` operation, SingularFS reassembles the `dirents` of the target directory by adopting prefix matching provided by the KV Store backend.

Data management. SingularFS employs a decoupled structure for metadata and data. Therefore, existing approaches for data management can be directly adopted by SingularFS. Currently, SingularFS uses the object store [31] to manage data. It indexes data blocks with key <`inode ID`+`block no`>.

3.2 Log-free Metadata Operations

In this section, we first analyze and classify the metadata write operations. Then, we demonstrate the timestamp dependency of the parent directory and child inodes and propose the core of log-free metadata operations, i.e., ordered metadata update.

3.2.1 Analysis of Metadata Write Operations

As illustrated in Table 1, metadata write operations in SingularFS can be classified into three categories according to the number of modified inodes.

1. *Single-node operations.* These operations only update the target inode itself (e.g., file `open/close/read/write`).
2. *Double-node operations.* These operations update the target inode, as well as the timestamps of its parent directory (e.g., file `create/delete`). Note that in SingularFS,

Operation Type	Metadata Write Operations	Modified Inodes		
		Target Inode	Parent Directory	Other Inodes
Single-node	`open/close/read/write/chmod/chown/utimens`	•		
Double-node	`mkdir/rmdir/create/delete`	•	•	
Rename	`rename`	•	•	•

Table 1: Classification of metadata write operations according to the modified inodes.

parent `dirents` are co-located with the child inode objects (§3.1), so there is no need to update the `dirents` separately for these operations.

3. *Rename operation.* This operation updates the original inode, the new inode, and their parent directories.

According to the real workloads shown in InfiniFS [19] and HopsFS [20], regular metadata operations, which include read operations, *single-node operations*, and *double-node operations*, account for more than 90% of all file system operations. Based on this observation, we design log-free metadata operations to guarantee the crash consistency of regular metadata operations.

Since the crash consistency of *single-node operations* can be directly guaranteed by the KV Store backend, in this section, we seek to guarantee the crash consistency of *double-node operations* without incurring additional overheads.

3.2.2 Ordered Metadata Update

As *double-node operations* set the parent directory's `ctime` to the target inode's *btime* or *dtime*, we observe that the timestamps of the parent directory and child inodes have the following dependency:

> *For an inode d, d.*`ctime` $\geq \max(c.btime, c.dtime)$*, where c is any of d's child inodes.*

Based on this observation, we update the metadata in order, to guarantee the crash consistency of *double-node operations* without incurring the logging overhead.

inode creation. As shown in Figure 4(a), inode creation includes two atomic steps. First, we insert the inode with *btime* = t_0, where t_0 is the current timestamp. Second, we set the `ctime` and `mtime` of its parent directory to t_0.

inode deletion. As shown in Figure 4(b), inode deletion includes three atomic steps. First, we invalidate the target inode and set its *dtime* to the current timestamp t_0. Second, we set the `ctime` and `mtime` of its parent directory to t_0. Finally, we physically remove the target inode from KV Store.

Crash recovery. When a crash occurs between the two steps in inode creation, or between step 1 and step 2 in inode deletion, the inconsistency can be identified by checking if the maximum *btime* and *dtime* of child inodes > `ctime` of the parent directory, and fixed by setting the `ctime` and `mtime` of the parent directory to the maximum value. When a crash occurs between step 2 and step 3 in inode deletion, the inconsistency can be identified by checking if the target inode is invalid and fixed by physically removing the invalid inode.

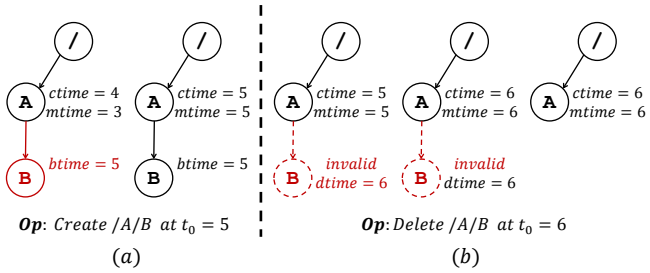


Figure 4: The process of inode creation and inode deletion in log-free metadata operations. (a) Inode creation includes two steps. First, insert the target inode with *btime* = current timestamp t_0. Then, update the timestamps of the parent directory to t_0. (b) Inode deletion includes three steps. First, mark the target inode as invalid and set its *dtime* to the current timestamp t_0. Then, update the timestamps of the parent directory to t_0. Finally, physically remove the target inode. The inconsistency of the directory tree is marked with red lines.

In order to detect and fix all the inconsistencies mentioned above, the most straightforward approach is to scan the whole directory tree. However, this process can be costly. SingularFS detects and fixes the inconsistency lazily when the inconsistent directory is accessed, as described in §4.

3.3 Hierarchical Concurrency Control

In traditional file systems, it is challenging to maximize the parallelism of *double-node operations* in a shared directory, as they cause contention over the parent `dirent` and timestamps. In SingularFS, the update of the parent `dirent` is co-located with the update of the child inode, whose concurrency is guaranteed by the KV Store backend. Therefore, the remaining challenge lies in the concurrent timestamp update.

We observe that *double-node operations* only modify the parent directory's `ctime` and `mtime`, whose size is 16B in total. Therefore, the concurrency control of the timestamp updates could be executed in a lock-free manner by leveraging the 16B atomic compare-and-swap instruction.

We divide operations on a target inode into three categories, *updater*, *writer*, and *reader*. *updater* contains inode update operations that involve only the target inode's `ctime` and `mtime`, and *writer* contains all other update operations. *reader* contains all inode read operations. For example, *double-node operations* like file `create/delete` are both *writers* of the target inode and *updaters* of the parent directory, and file `stat` is a *reader* of the target inode.

```
1  struct Inode {
2      ...
3      uint64 ctime; # aligned at a 16B boundary.
4      uint64 mtime;
5      ...
6  }
7
8  def writer(this: Inode *):
9      this->write_lock() # sync with other operations.
10     modify_inode(this)
11     this->write_unlock()
12
13 def updater(this: Inode *, timestamp: uint64):
14     this->read_lock() # sync with writer
15     while True:
16         # acquire timestamp snapshot.
17         cur = { this->ctime, this->mtime }
18         nxt = { timestamp, timestamp }
19         # update ctime and mtime atomically.
20         if (cur[0] >= timestamp or
21             cmpxchg16b(&this->ctime, cur, nxt)):
22             break
23     this->read_unlock()
24
25 def reader(this: Inode *):
26     this->read_lock() # sync with writer
27     while True:
28         # use ctime as the version number.
29         last_ctime = this->ctime
30         compiler_barrier()
31         inode = *this
32         compiler_barrier()
33         # OCC-like method.
34         cur_ctime = this->ctime
35         if cur_ctime == last_ctime:
36             break
37     this->read_unlock()
38     return inode
```

Algorithm 1: **Hierarchical concurrency control algorithm.**

Algorithm 1 shows the hierarchical concurrency control algorithm among *writer*, *updater*, and *reader*.

Writer-other synchronization. Synchronization between *writers* and other operations is handled by the per-inode read-write locks. The *writer* acquires the write lock to guarantee exclusive access to the target inode (line 9). *Updaters* and *readers* acquire the read lock to avoid concurrent *writer* doing inode update or remove operations (line 14, line 26).

Updater-updater synchronization. With the purpose of minimizing the length of the critical section, *updaters* use atomic instructions to synchronize between themselves in a lock-free manner. As the *updater* only updates the ctime and mtime of the target inode, they are placed in a 16B-aligned block (line 3), and the *updater* uses `cmpxchg16b` to atomically update `ctime` and `mtime` to the maximum of the timestamp parameter and the original value (lines 15-22). Specifically, the *updater* first acquires the current snapshot of `ctime` and `mtime` (line 17). If the current `ctime` is not less than the timestamp parameter, there is no need to update the timestamps as they have been updated by another *updater* with a later timestamp (line 20). If the current `ctime` is less than the timestamp parameter and there is no concurrent *updater* in the critical area (line 21), then update the timestamps atomically.

Updater-reader synchronization. We observe that `ctime` monotonically increases when *updaters* modify the inode, so it has the same semantic as a version number. Based on this observation, we adopt optimistic concurrency control (OCC) to synchronize between *readers* and *updaters* without locks. Specifically, `ctime` is fetched by the *reader* before and after getting the whole inode (line 29, line 34). The *reader* validates the inode by comparing the two `ctime`s (line 35).

3.4 Hybrid Inode Partition

In this section, we first show how to partition the inodes among NUMA nodes and execute multi-object directory operations introduced by the partition. Then, we propose the intra-NUMA data structure.

3.4.1 Inter-NUMA Inode Partition

Partition for NUMA locality of file operations. As file operations account for the majority of all metadata operations [19], we seek to guarantee their NUMA locality.

For *single-node* file operations, we delegate metadata requests to the corresponding NUMA node to ensure their NUMA locality. However, this does not work for *double-node* file operations (`create/delete`), as the two related inodes may reside in different NUMA nodes. Fortunately, in SingularFS, these operations only modify the parent directory's `ctime` and `mtime`, so NUMA locality of them can be guaranteed by grouping the parent directory's `ctime` and `mtime` with the child file inodes to the same NUMA node.

Therefore, SingularFS partitions the directory inode into timestamp metadata (`atime`, `ctime`, and `mtime`) and access metadata (inode ID, permission, *btime*, *dtime*, etc.). For each directory, SingularFS aggregates the directory timestamp metadata, its child file inodes, as well as its child directories' access metadata into a metadata group. The objects within a metadata group are placed in the same NUMA node. SingularFS uses consistent hashing to distribute the groups to NUMA nodes of the metadata server, thus achieving NUMA locality of file operations.

Multi-object directory operations. As the directory metadata is partitioned into two parts (i.e., two objects in the KV Store backend), the KV Store backend cannot intrinsically guarantee crash consistency for updates on the two parts. To solve this problem without incurring extra cost, SingularFS performs these operations in certain orders.

1. *Directory* ***`mkdir`*** *and* ***`rmdir`***. Three objects are involved in these operations, specifically the target directory's access metadata and timestamp metadata, as well as the parent directory's timestamp metadata. Note that when a directory is created, all its timestamps are set to its *btime*, which is inside the access metadata of the target directory. Leveraging this, SingularFS creates the target directory's timestamp metadata after creating its access metadata. After a crash, the target directory's timestamp metadata is re-generated with its access metadata.

2. *Directory* `set_permission`. Two objects are involved in this operation, specifically directory access metadata and timestamp metadata. To guarantee the crash consistency, we expand the semantic of *btime* to the maximum of born time and last `set_permission` time. When executing `set_permission`, SingularFS first updates the target directory's permission fields and its *btime* at the same time. Then, SingularFS updates its `ctime`. On crash recovery, SingularFS identifies whether the directory's *btime* is greater than its `ctime` and fixes them if needed.

3.4.2 Intra-NUMA Inode Partition

Partition for less lock contention. To generate `dirents` of a particular directory, SingularFS relies on the range query operation, which is supported by the ordered index. However, typical B+-tree-based ordered indexes suffer from high lock contention caused by traversing and node splits when updates prevail in workloads. To reduce such intra-NUMA lock contention while keeping the functionality of the range query, inside each NUMA node, SingularFS uses the hash algorithm to scatter all metadata uniformly to *n* ordered indexes, where *n* is a configurable parameter.

When executing point queries, SingularFS queries the value from the key's corresponding index calculated by its hash. When performing range queries, such as in directory `readdir` operation, SingularFS makes range queries in all the indexes and merges the results.

Optimization for removal. Directory `rmdir` requires determining whether the target directory is empty. As the `dirents` are co-located with the child inodes, this process can be implemented using prefix matching. However, with intra-NUMA inode partition, we would have to perform prefix matching in all *n* ordered indexes, which could be costly.

We optimize this process by adding a `num_dents` variable to the metadata of each directory, identifying the number of `dirents`. The crash consistency of this variable is easily guaranteed because it can be recovered by executing a complete prefix matching for the directory.

`num_dents` should meet two requirements for concurrency safety. First, it must allow concurrent updates from *updaters*. To achieve this goal, directory *updaters* use `fetch_and_add` and `fetch_and_sub` to synchronize with each other. Second, its value must be correct when a `rmdir` occurs. Since `rmdir` is a *writer* of the target inode, there can be no concurrent *updaters* when SingularFS is executing `rmdir`. Therefore, `num_dents` will keep unchanged during `rmdir` and it is safe for us to use its value to judge the directory's emptiness.

4 Crash Recovery

Algorithm 2 shows the overall crash recovery algorithm for a single directory inode in SingularFS. It mainly consists of three aspects: Parent timestamp recovery (lines 9-14) and

```
1  def recover(ino: Inode):
2      ino.num_dents = 0
3      # re-create incomplete timestamp metadata
4      recreate_timestamp_meta(ino)
5      # inconsistency caused by set_permission
6      if ino.btime > ino.ctime:
7          ino.ctime = ino.btime
8      for c in ino.child_inodes:
9          if (c.valid and c.btime > ino.ctime) or
10             (!c.valid and c.dtime > ino.ctime):
11             # crash before timestamp update.
12             ino.ctime = max(c.btime, c.dtime)
13             ino.mtime = max(c.btime, c.dtime)
14             redo(c) # create / delete.
15         else if not c.valid:
16             remove(c) # invalid inode
17         ino.num_dents += 1 # maintain num_dents
```

Algorithm 2: **Crash recovery algorithm for inode `ino`.**

invalid inode removal (lines 15-16) in log-free metadata operations, directory timestamp metadata recovery (lines 4-7) in inter-NUMA inode partition, and `num_dents` maintenance (line 2, line 17) in intra-NUMA inode partition.

SingularFS detects and fixes the inconsistent directory only when it is accessed. Specifically, SingularFS maintains a global `restartCnt` variable, which increases by one at each restart. Inside each directory inode, we keep a local `restartCnt` padded in the directory read-write lock. Locking a directory also sets its local `restartCnt` to the global value. After a crash, an inconsistent directory will contain an outdated `restartCnt`. Such inconsistency will be detected and fixed according to Algorithm 2 on the next access.

5 Evaluation

In this section, we evaluate SingularFS to answer the following questions:

- How does SingularFS compare to local PM file systems and distributed file systems utilizing RDMA and PM on metadata performance? (§5.2)
- How does SingularFS scale on concurrent racing metadata operations in a shared directory? (§5.3)
- How do the techniques employed by SingularFS impact its metadata performance? (§5.4)
- How does SingularFS perform with a billion-scale directory tree? (§5.5)
- What is the performance of `rename` in SingularFS? (§5.6)
- What is the end-to-end performance of SingularFS? (§5.7)
- What is the overhead of crash recovery? (§5.8)

5.1 Experimental Setup

Hardware configuration. In the experiments, unless otherwise stated, we use one server node and one or two client nodes. The metadata server and data server are co-located. Each server node has two Intel Xeon Gold 6330 CPUs, with 28 cores per socket. Hyperthreading is disabled on servers, for it aggravates contention and degrades the overall performance.

Each socket has four 128GB Intel Optane DC Persistent Memory (DCPMM) DIMMs, 256GB DRAM, and one 200Gb/s Mellanox ConnectX-6 NIC. The Optane DIMMs are configured in App Direct mode.

Each client node has two Intel Xeon Platinum 8360Y CPUs, with 36 cores (72 threads) per socket, so as to issue as many requests as possible to saturate the metadata server. Each socket has 256GB DRAM and one 200Gb/s Mellanox ConnectX-6 NIC. All the clients and servers are connected with a Mellanox QM8790 switch. For ConnectX-6 NICs, the driver and firmware versions are OFED 5.5-1.0.3.2 and 20.32.1010.

Compared systems. We use two types of baseline file system in the comparison, specifically local PM file systems and distributed file systems.

For local PM file systems, we choose Ext4-DAX and NOVA [32]. For these file systems, we clear the VFS cache before each directory and file `stat` operation to mitigate its impact and ensure that the acquired performance reflects the actual metadata performance of the file system.

For distributed file systems, we choose CephFS [31] and InfiniFS [19]. For CephFS, we use version 15.2.16 with RDMA enabled. The latency of CephFS is obtained by running it in RDMA mode, while the throughput is obtained by running it in IPoIB mode because we find that CephFS will always crash if run in RDMA mode for a relatively long time. For InfiniFS, we add support for RDMA and multiple NICs with eRPC [13]. Since InfiniFS uses RocksDB [5] storage backend, we run it on Ext4-DAX mounted on top of a RAID 0 device spanning across all the PM DIMMs.

We use P-Masstree [15] as the ordered index of SingularFS and set the per-NUMA index number n to 8. For SingularFS and InfiniFS, we use 56 worker threads executing in-line requests, and clients connect to them in a round-robin way. SingularFS stores its data objects in PM for fairness.

Benchmark. We use mdtest v3.3.0 provided by IOR [3] to evaluate the metadata performance of the aforementioned file systems. We use OpenMPI v4.1.2 to generate parallel mdtest client processes, which are placed on the metadata server for local PM file systems and scattered across all the client nodes for distributed file systems. We use the POSIX interface for local PM file systems. For distributed file systems, we use either the intercepted POSIX syscall [6] or the client libraries (e.g., libcephfs of CephFS). Our experiments create files of zero length like the previous works [17, 19, 25] because we focus on the insights into metadata performance. For end-to-end benchmarks, we use Filebench [27] to evaluate the overall performance.

For lack of multi-NUMA support in some of the compared file systems (e.g., NOVA and CephFS), we compare their per-NUMA throughput instead of overall throughput to guarantee the fairness of the comparison. For Ext4-DAX and NOVA, we limit their CPU and PM resources to a single NUMA node and use the results directly as their per-NUMA performance.

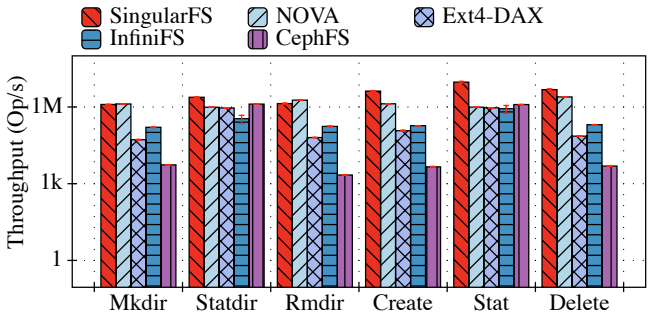


Figure 5: Per-NUMA throughput of metadata operations in private directories. The Y-axis is log-scaled.

For CephFS, we restrict its server-side PM and NIC resources and directly use the results as its per-NUMA performance. For SingularFS and InfiniFS, we do not limit their hardware resources and average their throughput results to get their per-NUMA performance.

5.2 Metadata Performance

In this section, we compare the overall metadata performance of the file systems mentioned above. We use mdtest to measure the performance of directory `mkdir/stat/rmdir` and file `create/stat/delete` operations. Each client handles 2 million directories and 2 million files in its private directory. For CephFS and InfiniFS, we get their peak performance by modestly reducing the directory and file quantity per client process, as they show a relatively low performance at the aforementioned scale.

5.2.1 Throughput

In this section, we evaluate the throughput of metadata operations in different file systems. We gradually increase the number of client processes to achieve the peak per-NUMA throughput for each file system.

Figure 5 shows the per-NUMA metadata throughput of different file systems in private directories. From the figure, we make the following observations:

1) SingularFS outperforms local PM file systems in file `create` and `delete` operations and outperforms the distributed file systems by more than an order of magnitude. SingularFS achieves 3.13×/1.93× and 22.49×/23.57× higher throughput for file `create`/`delete` operations than NOVA and InfiniFS. This is because the log-free metadata operations in SingularFS removes the extra transaction logic from the critical path of these operations, which saves both PM bandwidth and the CPU cycles spent writing logs and waiting for persistence. Inter-NUMA inode partition also guarantees NUMA locality of file operations, reducing inter-NUMA communication. These two designs make SingularFS fully exploit the single-server performance of file metadata write operations.

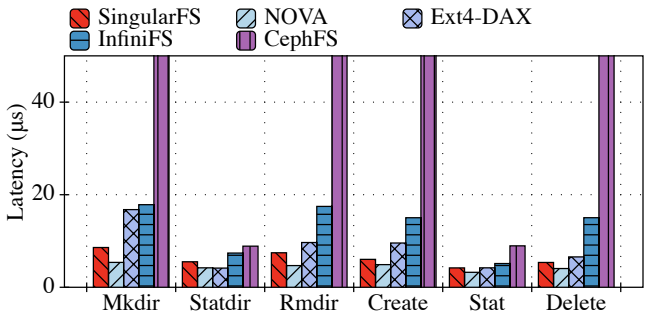


Figure 6: Latency of metadata operations. Note that the result of SingularFS, CephFS, and InfiniFS includes the network delay, while the result of Ext4-DAX and NOVA doesn't.

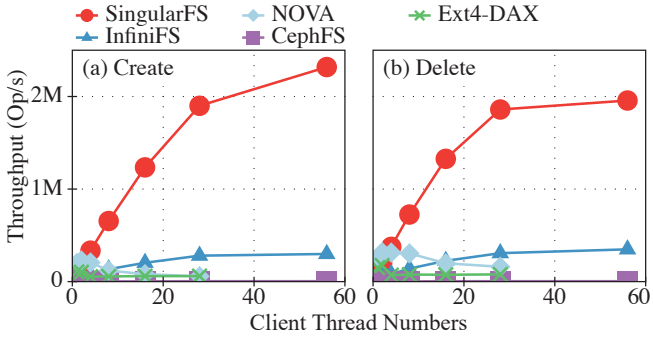


Figure 7: Per-NUMA throughput scalability of file `create` and `delete` in a shared directory.

2) The throughput of directory `mkdir` and `rmdir` is much lower than file `create` and `delete`, but still comparable with local PM file systems (0.96×/0.73× than NOVA) and much higher than distributed file systems (7.82×/7.77× higher than InfiniFS). This is because these operations need to write both the target directory's access metadata and its timestamp metadata, which are not guaranteed to be in the same NUMA node. However, these metadata update operations are transformed into several simple KV writes by utilizing log-free metadata operations, accelerating this process.

3) SingularFS outperforms local PM file systems and distributed file systems in the case of metadata read operations. For file `stat` operation, SingularFS achieves 9.54×/10.10× higher throughput than NOVA/Ext4-DAX, and 7.54×/10.97× higher throughput than CephFS/InfiniFS. This is because by adopting log-free metadata operations and hierarchical concurrency control, SingularFS separates the transaction logic from KV Store. This enables SingularFS to use a lightweight KV Store backend (e.g., P-Masstree [15]) to accelerate the `stat` operation.

4) SingularFS demonstrates high NUMA scalability. Using inter-NUMA inode partition, SingularFS guarantees NUMA locality for file operations. Although it utilizes all the hardware resources rather than just one NUMA node, its per-NUMA throughput of file operations still outperforms that of file systems running on one NUMA node.

5.2.2 Latency

In this section, we evaluate the latency of metadata operations in different file systems. We launch one mdtest client to measure the latency of each metadata operation.

Figure 6 shows the average latency of different metadata operations of the compared file systems. From this figure, we make the following observations:

1) Compared with local PM file systems, SingularFS achieves comparable latency with Ext4-DAX and NOVA in file operations. This is because SingularFS has a shorter server-side critical path for metadata write operations by leveraging log-free metadata operations and inter-NUMA inode partition. SingularFS also has a low latency for read operations by guaranteeing NUMA locality of file metadata. Even though suffering from the µs-level inherent latency of RDMA, SingularFS still achieves comparable latency with local PM file systems for file operations.

2) Compared with distributed file systems, SingularFS achieves lower latency than CephFS and InfiniFS. This is because with the lightweight KV Store and log-free metadata operations, SingularFS has a more lightweight software stack than CephFS and InfiniFS, contributing to its lower latency.

5.3 Scalability in a Shared Directory

In this section, we evaluate the throughput scalability of concurrent racing metadata operations in a shared directory. In the evaluation process, we gradually increase the number of mdtest clients and get the per-NUMA throughput for file `create` and `delete` operations in a shared directory.

Figure 7 shows the throughput scalability of file `create` and `delete` in a shared directory in different file systems. From the figure, we make the following observations:

1) SingularFS shows much better throughput scalability in a shared directory than other file systems, outperforming them by at least 7.76×/5.61× on file `create`/`delete` operations. This is because SingularFS leverages hierarchical concurrency control to maximize the parallelism of metadata operations in a shared directory, and adopts intra-NUMA inode partition to reduce lock contention inside the intra-NUMA data structure. These two methods contribute to the shared-directory scalability of SingularFS.

2) SingularFS achieves nearly the theoretical peak performance in a shared directory. The file `create`/`delete` throughput of SingularFS in a shared directory converges to around 4Mops/s for the whole metadata server, which is close to the per-NUMA peak throughput of SingularFS illustrated in Figure 5. This is because all the inodes in a shared directory are placed in the same NUMA node. Therefore, all the operations are handled in the same NUMA node, and the upper bound for performance is the per-NUMA peak performance of SingularFS.

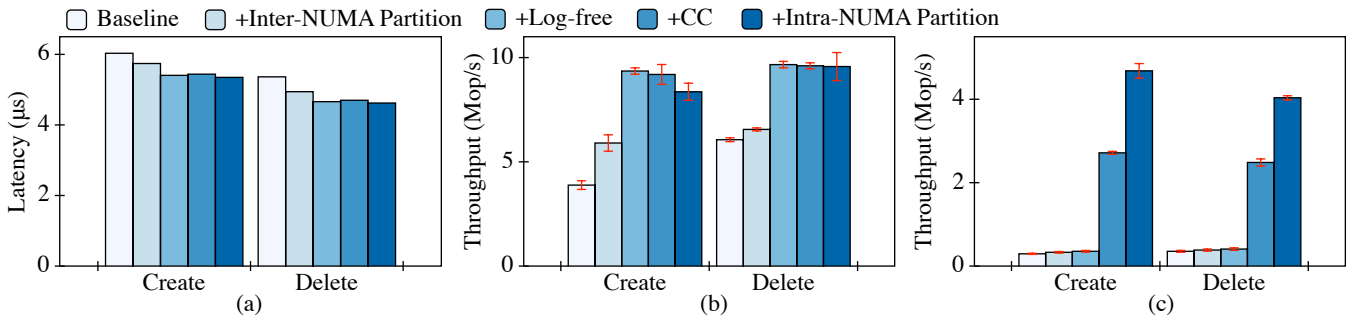


Figure 8: Performance breakdown. (a) Average latency. (b) Throughput in private directories. (c) Throughput in a shared directory. Design techniques are accumulated.

5.4 Factor Analysis

In this section, we analyze how our designs contribute to the latency and throughput by breaking down the performance gap between the *Baseline* and SingularFS. We apply our designs one by one to the *Baseline*, and measure the average latency and throughput of file `create` and `delete`. For the latency breakdown evaluation, we initiate one mdtest client to operate on 2 million files in a directory. For the throughput breakdown evaluation in private directories, we initiate 112 mdtest clients, and each client handles 2 million files in its private directory. For the throughput breakdown evaluation in a shared directory, we gradually increase the client number from 1 to 112 to achieve the peak throughput of each configuration. Each client operates on 2 million files in a shared directory. The results are not averaged to per-NUMA performance.

We implement the *Baseline* based on the framework of SingularFS, but without the key design features. It uses P-Masstree running on Ext4-DAX mounted on top of a RAID 0 device built from all the PM DIMMs, with pessimistic two-phase locking (2PL) and WAL for transaction support. In these three figures, *+Log-free* stands for utilizing log-free metadata operations instead of WAL to guarantee crash consistency. *+CC* represents adopting hierarchical concurrency control instead of simply using 2PL to do concurrency control. *+Inter-NUMA partition* and *+Intra-NUMA partition* are the two design parts of hybrid inode partition.

5.4.1 Private Directories

As shown in Figure 8(a) and Figure 8(b), SingularFS has much higher throughput and lower latency against *Baseline*. Specifically, for file `create` operation, SingularFS achieves 2.15× higher throughput with 1.13× lower average latency. For file `delete` operation, SingularFS achieves 1.58× higher throughput with 1.16× lower average latency. Here, we separately analyze all the design techniques in terms of file `create` workload (file `delete` has the same conclusions).

Inter-NUMA inode partition improves the throughput by 1.52× and reduces the average latency by 1.05×, since it guarantees NUMA locality of file operations, thus reducing the frequency of inter-NUMA PM access.

By using log-free metadata operations to guarantee crash consistency, SingularFS gains another 1.58× and 1.06× improvement in terms of throughput and average latency. This is because WAL is replaced in the critical path with log-free metadata operations, reducing the bandwidth waste of PM for logs and saving the CPU cycles of log persistence.

Hierarchical concurrency control and intra-NUMA partition bring a 10% throughput drop and a minor latency increase since they introduce extra synchronization overhead when there are few conflicts. Besides, the intra-NUMA inode partition has a negative effect on the cache friendliness of the overall data structure, accounting for the throughput drop.

5.4.2 Shared Directory

Figure 8(c) shows the throughput of file `create` and file `delete` in a shared directory. Compared with *Baseline*, SingularFS achieves 15.93×/11.40× higher throughput for file `create/delete` operations separately.

Inter-NUMA inode partition and log-free metadata operations contribute to 1.21×/1.15× throughput increase for file `create/delete` operations by reducing the length of the critical area, which shortens the critical area. However, the major bottleneck still lies in the lock contention of the shared directory. Hierarchical concurrency control mitigates this lock contention, thus increasing the throughput by 7.66×/6.09×. This is the peak throughput of P-Masstree in the case of high lock contention brought by the common prefix. The intra-NUMA inode partition mitigates this lock contention and contributes to another 1.72×/1.62× higher throughput.

5.5 Billion-scale Directory Tree

In this section, we demonstrate that SingularFS can efficiently support the billion-scale directory tree. We repeatedly `create` and `stat` 50 million files per NUMA node to increase the directory tree size until the file system is full.

Figure 9 shows the evaluation results. We make the following observations from the figure:

1) SingularFS delivers a steadily high throughput for file `create` and file `stat` operations with the billion-scale direc-

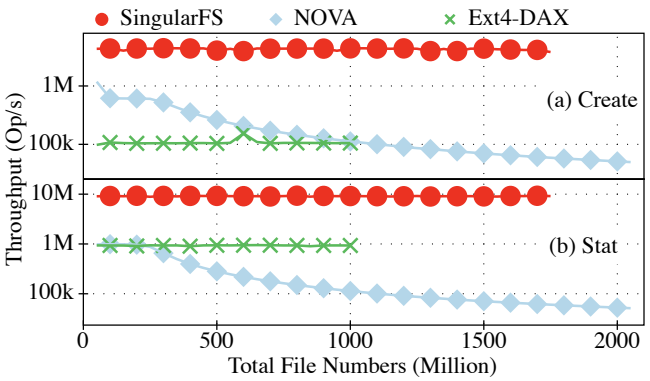


Figure 9: Per-NUMA throughput of file `create` and `stat` for the billion-scale directory tree. The Y-axis is log-scaled.

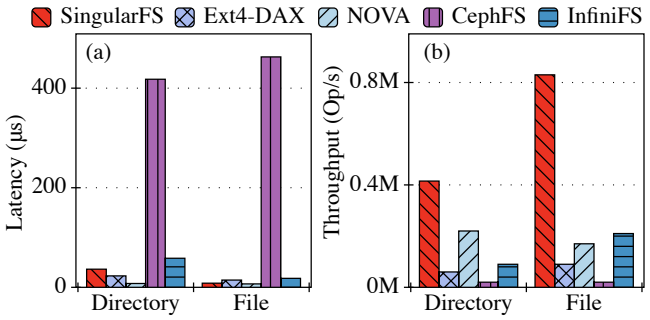


Figure 10: Performance of directory `rename` and file `rename`. (a) Latency. (b) Per-NUMA throughput.

tory tree. Specifically, it shows a steady per-NUMA throughput of ~4.40Mops/s for file `create` and ~9.10Mops/s for file `stat`, which is 41.90×/9.83× higher than Ext4-DAX. Although NOVA shows a relatively high throughput when the directory tree is nearly empty, its throughput is not even as high as Ext4-DAX when there is a billion-scale directory tree.

2) SingularFS shows similar directory tree scalability to local PM file systems. SingularFS supports 1.75 billion files per NUMA node, which is higher than Ext4-DAX (1 billion files) and lower than NOVA (2.05 billion files). The reason why SingularFS supports fewer files than NOVA is that SingularFS uses a KV Store to store the inodes, while NOVA uses per-core linked lists. The index of SingularFS has a higher capacity overhead than NOVA. However, it provides significantly higher performance.

5.6 Rename

In this section, we evaluate the latency and per-NUMA throughput of directory `rename` and file `rename` in SingularFS. In the experiments, each client renames 1 million directories and 1 million files from one directory to another.

Figure 10 shows the overall results. From the figure, we make the following observations:

1) For file `rename`, SingularFS shows comparable latency

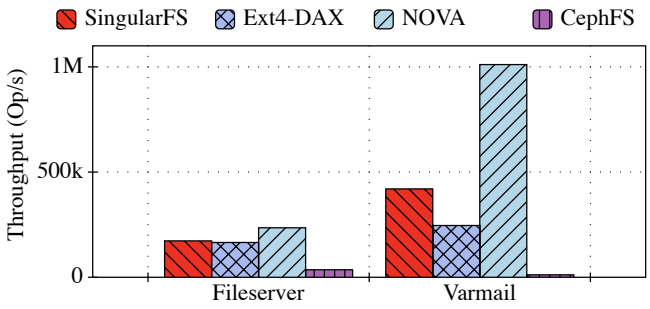


Figure 11: Per-NUMA throughput of Fileserver and Varmail workloads in Filebench.

with local PM file systems and at least 3.95× higher per-NUMA throughput than other file systems. Although SingularFS uses journaling to ensure crash consistency, the journal is lightweight as it only contains the two target KV pairs. Logs of the parent directories are omitted with log-free metadata operations. However, because of the journaling overhead and inevitable inter-NUMA access, the throughput of file `rename` is 0.17×/0.15× of file `create`/`delete`, lower than a half of their throughput (i.e., theoretical throughput limit).

2) The latency and throughput of directory `rename` are both worse than those of file `rename`. This is because SingularFS adopts the directory metadata cache like the previous works [17, 19, 25]. Directory `rename` causes cache invalidation and path re-resolution, reducing its performance.

5.7 End-to-end Performance

In this section, we test the end-to-end performance of SingularFS. Specifically, we run the per-NUMA throughput of Filebench Fileserver and Varmail workloads on SingularFS and baseline systems. We set the file number of Varmail to 100K. The result of InfiniFS is not included as it focuses on metadata service rather than the whole file system.

Figure 11 shows the results. From the figure, we make the following observations:

1) SingularFS outperforms CephFS and Ext4-DAX in both workloads. Specifically, SingularFS outperforms Ext4-DAX by 1.05× in Fileserver and 1.71× in Varmail. With its metadata design, SingularFS gets more performance gains in Varmail, which is metadata-intensive.

2) SingularFS shows 0.74× and 0.41× the throughput of NOVA in Fileserver and Varmail respectively. The lower performance of SingularFS stems from the existing inter-client metadata dependencies (e.g., client 1 creates a file, then client 2 deletes it) within the workload, which makes the high performance of the metadata server hard to be fully exploited. As the metadata server is under-saturated, the latency of metadata operation becomes the primary influence on the overall performance. Because SingularFS is accessed over the network, it has higher metadata operation latency (as discussed in §5.2.2) and thus lower throughput than NOVA.

Method	*Launch*	*Consistent*	*Inconsistent*
Scan	32.1s	5.3µs	-
Lazy	0s	5.3µs	30.8ms

Table 2: Recovery overhead. *Launch*: recovery time during launch. *Consistent*: latency of file `create` that does not require recovery. *Inconsistent*: latency for file `create` to detect and fix the inconsistent parent directory.

5.8 Crash Recovery

In this section, we evaluate the impact of lazy recovery on the latency of metadata operations, compared to scanning the directory tree. In the experiments, the server crash results in one inconsistent directory with 100K files in it. For lazy recovery, SingularFS detects and fixes this inconsistent directory while doing file `create` in it. For scan, the server scans the whole directory tree during launch. The directory tree size is 100M.

Table 2 shows the results. From the table, we make the following observations:

1) The time of lazy recovery is much shorter than scanning the directory tree. This is because SingularFS only scans the 100K files in the inconsistent directory on lazy recovery. However, all the 100M files must be accessed when scanning the whole directory tree, contributing to its high latency. This issue is even more severe in the billion-scale directory tree.

2) The lazy recovery time is still orders of magnitude higher than metadata operations. This is because SingularFS still needs to scan the 100K files in the inconsistent directory during lazy recovery. The recovery time is expected to be lower if the inconsistent directory contains fewer files.

6 Related Work

The efficiency of file systems has always been an interesting and important research topic, both for local PM file systems [10, 11, 32, 35] and distributed file systems [7, 17, 19, 22, 25, 31]. Different from the works mentioned above, SingularFS improves metadata efficiency by optimizing the transactions in metadata operations with ordered updates and improving NUMA locality of metadata operations. In this section, we focus on these two aspects of related work.

Transactions and Ordering in metadata operations. Transactions are a way to guarantee the atomicity and crash consistency of metadata operations. For local PM file systems, BPFS [10] relies on copy-on-write (CoW) and 8-byte in-place atomic updates to provide metadata consistency. PMFS [11], in contrast, uses larger in-place atomic updates with journaling, while NOVA [32] uses a log-structured data structure for metadata consistency. For distributed file systems, HopsFS [20] relies on both row-level locking and the NDB backend for strong metadata consistency semantics, while InfiniFS [19] leverages the transaction mechanism of the key-value storage backend and the two-phase commit protocol separately for local and distributed metadata transactions. CephFS [7, 31] and CFS [29] also leverage their storage backend to guarantee the atomicity and crash consistency of metadata operations. These methods have considerable CPU overhead, and SingularFS mitigates this overhead by using log-free metadata operations to guarantee crash consistency with minimal cost and adopting hierarchical concurrency control to maximize parallelism.

Ordering is another way to support crash consistency. Soft Updates [12] ensures that data and metadata are written to disks in an ordered way, so as to enable recovery after a crash. One of the obstacles to using Soft Updates is the complexity of keeping general orders between different metadata blocks. In comparison, SingularFS only needs to keep the orders of timestamp metadata, and thus is more practical in use.

NUMA-aware file systems. Several recent studies propose approaches for mitigating the NUMA effect in PM file systems. NThread [30] uses thread migration to alleviate the NUMA issues of the PM file systems. Assise [8] uses on-die DMA engines for remote PM writes to bypass hardware cache coherence. OdinFS [35] uses NUMA-aware delegation threads to handle PM access with large granularity. These approaches are mainly for data service in the file system, and can not be easily applied to metadata service, because metadata operations 1) are not more costly than thread migration, 2) have a more complex indexing logic other than simple read/write, 3) have small access granularity. SingularFS uses hybrid inode partition to ensure NUMA locality of file operations while minimizing the extra cost of scheduling.

7 Conclusion

This paper presents SingularFS, a billion-scale distributed file system using a single metadata server. SingularFS uses log-free metadata operations to eliminate additional crash consistency overheads for most metadata operations; then uses hierarchical concurrency control to maximize the parallelism of metadata operations; and finally, takes hybrid inode partition to reduce inter-NUMA access and intra-NUMA lock contention. Our extensive evaluation shows that SingularFS consistently provides high performance for metadata operations on both private and shared directories, and has a steadily high throughput for the billion-scale directory tree.

Acknowledgments

We sincerely thank our shepherd Emmett Witchel for helping us improve the paper. We are also grateful to the anonymous reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. 62022051, 61832011, U22B2023), and China Postdoctoral Science Foundation (Grant No. 2022M721828).

References

[1] one usage of up to a million files/directory. https://leaf.dragonflybsd.org/mailarchive/kernel/2008-11/msg00070.html, 2008.

[2] Intel® Optane™ Persistent Memory 200 Series Brief. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/achieve-greater-insight-with-intel-optane-persistent-memory-brief.pdf, 2021.

[3] HPC IO Benchmark Repository. https://github.com/hpc/ior, 2022.

[4] NVIDIA Mellanox ConnectX-6 SmartNIC Adapter. https://www.nvidia.com/en-us/networking/ethernet/connectx-6/, 2022.

[5] RocksDB. http://rocksdb.org/, 2022.

[6] syscall_intercept. https://github.com/pmem/syscall_intercept, 2022.

[7] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-Local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[9] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery.

[10] 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 ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[11] 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 Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[12] Gregory R Ganger, Marshall Kirk McKusick, Craig AN Soules, and Yale N Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[13] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs Can Be General and Fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 1–16, USA, 2019. USENIX Association.

[14] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. Z-Journal: Scalable Per-Core Journaling. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 893–906, 2021.

[15] 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 '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 213–226, USA, 2008. USENIX Association.

[17] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. LocoFS: A Loosely-Coupled Metadata Service for Distributed File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write Dependency Disentanglement with HORAE. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[19] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, Santa Clara, CA, February 2022. USENIX Association.

[20] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 89–103, USA, 2017. USENIX Association.

[21] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798, 2017.

[22] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 13, USA, 2011. USENIX Association.

[23] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is Turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 71–85, 2022.

[24] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, 2013.

[25] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 237–248. IEEE Press, 2014.

[26] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, page 4, USA, 2000. USENIX Association.

[27] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[28] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111, 2021.

[29] Yiduo Wang, Yufei Wu, Cheng Li, Pengfei Zheng, Biao Cao, Yan Sun, Fei Zhou, Yinlong Xu, Yao Wang, and Guangjun Xie. CFS: Scaling Metadata Service for Distributed File System via Pruned Scope of Critical Sections. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 331–346, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Ying Wang, Dejun Jiang, and Jin Xiong. Numa-aware thread migration for high performance nvmm file systems. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology*, 2020.

[31] 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*, OSDI '06, page 307–320, USA, 2006. USENIX Association.

[32] 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'16, page 323–338, USA, 2016. USENIX Association.

[33] Bin Yang, Wei Xue, Tianyu Zhang, Shichao Liu, Xiaosong Ma, Xiyang Wang, and Weiguo Liu. End-to-End I/O Monitoring on Leading Supercomputers. *ACM Trans. Storage*, nov 2022. Just Accepted.

[34] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. HTMFS: Strong Consistency Comes for Free with Hardware Transactional Memory in Persistent Memory File Systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 17–34, Santa Clara, CA, February 2022. USENIX Association.

[35] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, 2022.

The Hitchhiker's Guide to Operating Systems

Yanyan Jiang
Nanjing University

Abstract

This paper presents a principled approach to operating system teaching that complements the existing practices. Our methodology takes state transition systems as first-class citizens in operating systems teaching and demonstrates how to effectively convey non-trivial research systems to junior OS learners within this framework. This paper also presents the design and implementation of a minimal operating system model with nine system calls covering process-based isolation, thread-based concurrency, and crash consistency, with a model checker and interactive state space explorer for exhaustively examining all possible system behaviors.

1 Introduction

> "Everything should be made as simple as possible, but no simpler." —*Albert Einstein*

The teaching foundation of operating system design and implementation has been well-established for decades. From Tanenbaum's "*Operating Systems: Design and Implementation* (1987)" [45] to Arpaci-Dusseau's "*Operating Systems: Three Easy Pieces* (2018)" [3], students approach operating systems by studying the layered design of abstractions over processors, memory, and storage systems.

In parallel, researchers have observed the emergence of fast, scalable, reliable, and secure systems over the past few decades. This progress has been driven by the development of innovative system technologies, such as hardware/software co-design [24, 43], cross-stack integration [20, 23], program analysis [11, 48], and formal methods [30, 31], among others.

This paper attempts to share these exciting ideas with junior operating system learners under a unified theme by "adding a layer of indirection." Our key insight is to view all components of a computer system–including hardware, applications, and the operating systems that connect them–as state transition systems. By analyzing these components as informal yet mathematically rigorous objects, we aim to bridge the gap between theoretical concepts and practical system implementations.

This model-driven approach is grounded in several innovative philosophies on operating systems education, which are outlined below:

Everything is a state machine (Section 2). The key idea of this paper is to consider state transition systems as the foremost concept in teaching operating systems. The state-machine abstraction is fundamental: the state of a modern multi-processor system is essentially determined by register/memory bit values, driven by the non-deterministic selection of a single CPU executing a single-step instruction[1]. The same abstraction is also applicable to any multi-threaded program.

Consequently, we argue that it is beneficial to *view the operating system as both a state machine and a manager of state machines*. An operating system essentially leverages application-invisible data structures (e.g., a page table) to multiplex CPUs across processes and threads. This approach provides a rigorous explanation of process management APIs: `fork`/`execve`/`exit` functions simply clone, reset, and destroy live state machines. This abstraction also encourages in-depth discussions about `fork` [4] and the initial process state after `execve`.

By adopting this state-machine-centric perspective, we can explain research systems with a clear and rigorous foundation. For instance, every debugger [12], trace [8], and profiler [9] essentially "observes" runtime state snapshots, facilitating discussions on interactive query debuggers [39], deterministic full-system replay [16], time-travel failure reproduction [11], snapshot-based fault tolerance [38], and state space exploration [7] in an introductory-level operating system course.

Emulate state machines with executable models (Section 3). Since state machine is a mathematically rigorous concept, we could always *emulate* the execution of real state machines. Although emulation has been widely adopted in operating sys-

[1] Under the assumption of race-freedom that instructions are serializable.

System Call	Description
fork()	Create current thread's heap and context clone
spawn(f,xs)	Spawn a heap-sharing thread executing $f(xs)$
sched()	Switch to a non-deterministic thread
choose(xs)	Return a non-deterministic choice among xs
write(xs)	Write strings xs to standard output
bread(k)	Return the value of block k
bwrite(k,v)	Write block k with value v to a buffer
sync()	Persist all outstanding block writes to storage
crash()	Simulate a non-deterministic system crash

Table 1: System calls in the operating system model.

tem teaching[2], this paper takes one step further by emulating a "fully functional" operating system model with processes, threads, a debug console, and block storage. The system calls are listed in Table 1. The executable model approach has the following advantages:

First, *executable model is a foundation for exploring operating system concepts*. Synchronization primitives like Peterson's algorithm [34], condition variable, and semaphore can be implemented over shared memory. The non-trivial state copy behavior of `fork()` [4] can be reproduced under this model. A file system checker can be carried out upon a simulated `crash()`.

Second, *executable model is a behavioral specification of real operating systems*; it is the golden standard on the application-observable behaviors. A model facilitates discussions on the abstractions–the concrete implementation of the `fork()` function may employ copy-on-write, but this should remain transparent to a process. Such a model also motivates the key idea behind formally verified systems like seL4 [25] and Hyperkernel [31].

Enumeration demystifies operating systems (Section 4). We design our emulator to handle all sources of non-determinism in a coherent way: every system call (not merely `choose`) returns a set of possible choices as callbacks. Consequently, we can exhaustively enumerate all possible system behaviors with little implementation effort.

Such a design finally leads to our MOSAIC (Modeled Operating System And Interactive Checker) *operating system model and checker*. MOSAIC adds lightweight formal methods [21,47] to operating systems teaching. MOSAIC is capable of checking fork-based process parallelism, thread-based shared memory concurrency, and crash consistency [36]. The model checker's output can be piped to an interactive state space explorer that can be embedded in a Jupyter notebook (Figure 3); thus, all non-trivial corner cases of the operating system model can be rigorously explained.

In summary, this paper makes the following contributions:

1. We propose a new "state-machine first" approach in the breakdown of operating system teaching: (1) model systems as state machines, (2) realize models by emulation, and (3) explore models by enumeration. This approach enabled us to introduce non-trivial research systems to junior operating system learners.

2. We design and implement MOSAIC, a minimal (500 lines of code, including comments) executable operating system model and checker, which strikes a balance between understandability and functionality. MOSAIC can rigorously explain non-trivial textbook cases concerning concurrency, virtualization, and persistence. MOSAIC is available via

   `https://github.com/jiangyy/mosaic`.

3. We incorporated these ideas in a first undergraduate operating system course (Section 5). This course became one of the most popular operating system courses in China and has attracted over 2,000,000 video views since its initial release in 2020.

2 State Machines: First-class Citizens of Operating Systems

Philosophy 1: Everything is a state machine.

This paper's key contribution is the "state-machine first" approach to operating systems. By regarding both user-level applications and kernels as state machines (Section 2.1), it became obvious that operating systems are state machine managers (Section 2.2). This section also discusses modern computer systems and tools under the state machine perspective (Section 2.3).

2.1 Introducing State Machines in the Operating System Class

Program as a state machine. Every program run essentially boils down to the execution of binary instructions, whose behavior is rigorously defined by a state machine in which states are register/memory values and transitions are the execution of one instruction at the program counter. We *implement* this idea on Linux (Figure 1) to provide a definition of system calls: system call is a state transition (e.g., via a trap instruction or any process-kernel communication mechanism [44]) for accessing the "exterior" of the state machine, e.g., writing data to the operating system or changing the state machine's memory address space (via `mmap` or `mprotect`) and existence (via `exit`). Without system calls, the program (state machine) is a "closed world" that can only perform arithmetic and logical operations over memory and register values.

[2]We loved the emulated process scheduler, virtual memory, and file systems in the "Three Easy Pieces" [3].

```
1  #include "sys/syscall.h"
2  mov $SYS_write, %rax  // write(
3  mov $1,         %rdi  //   fd=1,
4  mov $hello,     %rsi  //   buf=hello,
5  mov $16,        %rdx  //   count=16
6  syscall               // );
7  // "ret" here yields SIGSEGV
8  mov $SYS_exit,  %rax  // exit(
9  mov $1,         %rdi  //   status=1
10 syscall               // );
11 hello: ; .ascii "Hello, OS World\n"
```

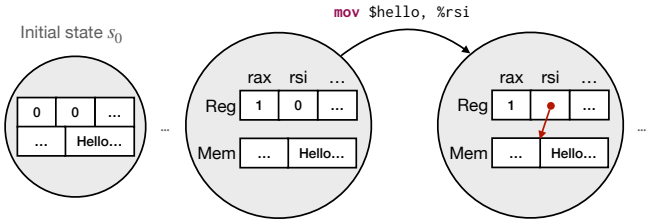


Figure 1: A minimal "Hello World" program and its corresponding state machine.

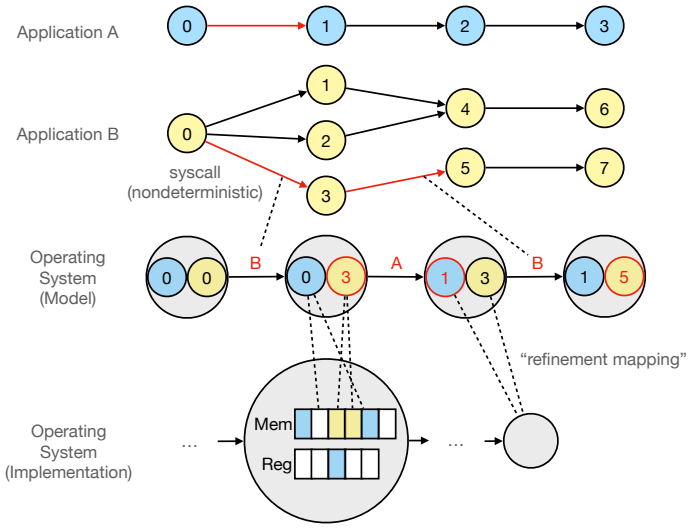


Figure 2: Operating system as a state machine manager. In this example, the operating system "executes" state transitions $0 \to 1$ and $0 \to 3 \to 5$ for applications A and B, respectively.

Bare-metal as a state machine. The bare-metal hardware shares a similar model with binaries: a CPU essentially operates as an infinite loop of instruction execution, which is also the case for a full-system emulator [5]. In contrast to user-level programs that can perform system calls, bare-metal kernels (including operating systems) access the "external world" via port or memory-mapped I/O and can be interrupted as if a trap instruction is non-deterministically injected.

Discussions. The advantage of introducing the state machine model early in an operating system course is that it fosters a tendency of *rigorous thinking*–state transition systems are well-defined mathematical objects. Specifically, we motivate the students to think of what is the mathematically precise definition of the process initial state. We explain that any process's initial state is well-defined by its binary executable and the Application Binary Interface. We also demonstrate how to inspect the initial state of the code in Figure 1 using `stepi` in GDB and memory mapping files in procfs. We further encourage students to consider more involved details of process states, such as the reasons behind the inability to perform a function return (using a `ret` instruction) and the necessity of wrapping C `main` functions with a `__libc_start_main`.

2.2 Operating System as a State Machine Manager

Computer system stack on state machines. Virtualization is the most fundamental mechanism of modern operating systems. Each application in an operating system can be regarded as a state machine whose initial memory layout and state transitions are specified by its binary executable. The operating system should give the application the illusion that the state transition system runs continuously following its specification, even though instruction execution could be non-deterministically interrupted at any time.

The state-machine approach provides a natural "implementation" of virtualization: by making state snapshots of all processes available and scheduling a process through "moving" a state machine to the CPU. The trap/interrupt handler plays such a role: it stores the state machine's registers in the operating system's private memory space, ensuring the system-wide invariant that all application states can be reconstructed. Subsequently, the operating system can continue processing interrupts, executing system calls, and resuming any process based on a predefined scheduling policy.

These arguments conclude our claim that "everything is a state machine" and gives us a new picture of understanding operating systems, as shown in Figure 2:

1. Application code is the developer's specification of a state machine.

2. Operating system code is the designer's specification of a state machine manager, a "superset" state machine container of all application state machines.

3. The operating system provides system calls as services and leverages application-invisible states (e.g., page table) to give processes the illusion of continuous state machine execution.

Process APIs on state machines. Following the idea that running applications are state machines, the need for process APIs became obvious: an operating system must provide

mechanisms for manipulating the set of live state machines. We found that the state machine language[3] precisely and concisely explains UNIX process APIs:

1. `fork()` makes a "full copy" of the currently running state machine. Registers and the address space should appear to be deeply copied. References to operating system objects (e.g., file descriptors and signal handlers) should also be copied, but with caution [4].

2. `posix_spawn(...)` creates a new state machine (always resets to the initial state of an application) with controllable state sharing with the parent.

3. `execve(path, argv, envp)` resets a running state machine to the initial state specified by the binary file `path`, with arguments `argv` and environment list `envp` placed in memory following the Application Binary Interface.

4. `exit(status)` removes the currently running state machine from the operating system, reclaims used resources, and notifies any waiting process with the exit `status`.

2.3 State Machines Meet Operating Systems

We discovered that the state-machine approach is not only beneficial for clarifying operating system concepts, but it can also serve as a fundamental basis for explaining non-trivial research systems to students:

Understanding system execution. Theoretically, executing a state transition system (be it an application or an operating system kernel) results in an execution trace composed of state snapshots connected by state transitions

$$tr = s_0 \to s_i \to \ldots \to s_{i+1} \to \ldots,$$

as if we single-instruction debug the program and save a core dump after each instruction execution. Such a trace contains *all* information needed for understanding this specific program execution.

However, such a massive trace (billions of instructions executed per second and megabytes of snapshots) is impractical and unnecessary to keep for any engineering practice. Debuggers provide the break/watchpoint mechanism to efficiently stop at interested program points (sometimes with hardware assistance like debug registers) and let the developer examine the program states interactively.

Understanding a program's execution usually only requires a tiny fraction of information in the full trace *tr*. The trade-off space of "what parts of *tr* to observe" leads to many important mechanisms incorporated in the engineering of modern operating systems, which are explained below.

[3]For brevity, we removed less critical mechanisms including signals, process groups, and access control in this discussion. However, all of them can be explained under the state machine perspective whenever needed.

Playing with snapshots. `fork` provides a verbatim copy of a program's state s_i with reasonably low cost. Holding such program state snapshots yields interesting applications. One is the Zygote process of Android [14], which copies initialized Java virtual machine state to avoid repetitive and time-consuming bootstrap-time class loading. Another example is that one can take periodical clean-state snapshots (e.g., in the idle state of an event loop) and fall back to a snapshot when an unexpected error occurs [38].

Time-travel debugging. Developers use a debugger to interactively examine *tr*, which can be enhanced by a query language [39]. Debuggers can also enable time-travel debugging by recording the differences between consecutive states, essentially creating an undo log. Time-travel debugging is already implemented in GDB [12]. Observing that non-deterministic transitions are only a tiny fraction of *tr*, one can also keep track of their locations and choices to enable a deterministic replay [16, 33].

Trace and profiler. One can insert probes exclusively at state transitions relevant to the application logic (e.g., function calls and returns) and gather diagnostic data (e.g., call stack traces). Trace utilities such as ftrace and Kprobe in Linux [8] are widely used for debugging production failures.

One can place probes only at application logic relevant state transitions (e.g., function calls and returns) to collect diagnostic information (e.g., call stack trace). Such trace tools like ftrace and Kprobe in Linux [8] are widely used in debugging production failures.

The overhead associated with tracing can be further reduced through sampling, which involves periodically activating probes within a specified time interval. Such profilers generate summaries of the sampled program states and are extremely useful in diagnosing performance issues.

Runtime checkers. Runtime checkers can also be considered as functions that accept *tr* as input and check it against specific bug patterns. A broad spectrum of checkers operate in this manner: AddressSanitizer [40] asserts the absence of out-of-bounds and use-after-free memory accesses. ThreadSanitizer [41] confirms that there are no conflicting shared memory accesses unordered by happens-before relations. Lockdep [29] checks whether all observed lock acquisition orderings do not form a cycle.

Symbolic execution and program verification. It is obvious that system calls can exhibit non-deterministic behavior. However, it is less emphasized that such non-determinism can be rigorously quantified; for instance, a read system call returns only a finite number of possibilities. Thus, we can enumerate all possible state transitions to capture all potential program behaviors; however, this approach is only feasible in a theoretical context. Even reading a 32-bit integer results in 2^{32} distinct states.

Using a compact representation of a vast number of states (e.g., using a symbolic value x to represent an "arbitrary"

value of variable `x`) and imposing constraints on symbolic values across branches results in a symbolic program verifier [7].

3 An Executable Operating System Model

> **Philosophy 2**: Emulate state machines with executable models.

As state machines are mathematically rigorous constructs, their usefulness is not limited to merely clarifying operating system concepts. It is also feasible to develop *executable* state machines that accurately emulate the behavior of processes and operating systems.

Specifically, we leverage modern programming language mechanisms like coroutines for lightweight in-process context switches to implement a lightweight executable operating system model with emulated threads, processes, and devices (Section 3.1). This section also discusses how instructors could use a model to simplify non-trivial textbook cases (Section 3.2) and use models as behavioral specifications of real systems (Section 3.3).

3.1 Emulating an Operating System

State machines (processes) and system calls. We implement our operating system model in Python, a popular programming language among students. A process is emulated by a generator (stackless coroutine) object where process memory is its local variables. System calls (Table 1) are emulated by `yield` in which the generator saves its local state (local variables and program counter) in a closure and transfers control to its caller[4]:

```
def main(msg):  # an emulated application process
  i = 0
  while (i := i + 1):
    yield 'SYS_write', msg, i  # write(msg, i)
    yield 'SYS_sched',  # sched()
```

Our operating system model, as a state machine manager, maintains a set of processes (continuable generators) and is an infinite loop of `yield` trap handler, just like any real operating system:

```
class OperatingSystem:
  def __init__(self, procs):  # OS initialization
    self._procs = procs
    self._current = procs[0]

  def run(self):  # the OS main loop
    while True:
      syscall, *args = self._current.__next__()
      match syscall:
        case 'SYS_write':  # write to debug console
          print(*args)
        case 'SYS_sched':  # switch to a random process
          self._current = random.choice(self._procs)

OperatingSystem([main('ping'), main('pong')]).run()
```

[4] In the MOSAIC implementation, the process code is stored in a standalone Python file. Applications invoke system calls in Table 1 as ordinary function calls like `x = sys_choose(['Head', 'Tail'])`, and MOSAIC rewrites the AST by replacing all system call nodes to `yield`.

Process APIs. Because deep-copying a generator object is not allowed in Python, we implement `fork()` by creating a new `OperatingSystem` object and replaying all executed system calls to obtain a deep copy of the process. This requires `OperatingSystem` to keep track of the non-deterministic choices of all previously executed system calls. Processes have increasing IDs starting from 1,000, and the child process ID is returned on `fork()`. There is no `exit()` because returned generators are never scheduled and are considered exited. There is also no `execve()` because its functionality largely overlaps with `spawn()` and `fork()`.

Threads and shared memory. The shared memory among threads is emulated by the global `heap` variable, whose value is updated before switching to a process/thread by

```
globals()['heap'] = self._current.heap,
```

and readers may notice that this `heap` models a "page table base register" which is changed on context switches. `spawn(f, *xs)` creates a new generator calling `f` with arguments `xs` and a shared `heap`. The replay-based `fork()` obtains a deep copy of the heap in the freshly allocated `OperatingSystem` object.

Devices. Writing to the debug console appends the message to a buffer. Reading from the debug console can be implemented by `choose()` from possible inputs. The emulated block device is a key-value mapping, which maps each block's ID (any string like `inode` or even emojis) to its contents (any serializable data structure including strings and lists). All block device writes are first appended to a queue to simulate real disks with a volatile buffer [36]. Write-back happens only when `sync()` is called.

3.2 Modeling Operating System Concepts

Such a surprisingly simple model can simplify textbook cases that require non-trivial interactions across system layers and are thus challenging to debug or even reproduce–we can selectively model the essential elements of the system to minimize the complexity:

A `fork()` in the road [4]. Fork is no longer simple, considering it conducts a full state copy of libraries and references (handles) to operating system objects. Below is such a non-trivial case related to the buffer mechanism in the standard C libraries:

```
for (int i = 0; i < 2; i++) {
  int pid = fork();
```

```
  printf("%d\n", pid);
}
```

```
(unix) $ ./a.out
1000
1001
0
0
1002
0
```

```
(unix) $ ./a.out | wc -l
8  # ???
```

Debugging the internal implementation of libc (even with a much simpler implementation like musl [1]) to understand this case requires substantial engineering efforts. Alternatively, we first model this case by removing all low-level details of process creation and focusing on the behavior of a fork-cloned buffer:

```
def main():
  heap.buf = ''
  for _ in range(2):
    pid = sys_fork()  # heap.buf is deeply copied
    sys_sched()  # non-deterministic context switch
    heap.buf += f'{pid}\n'  # or sys_write()
  sys_write(heap.buf)  # flush buffer at exit
```

The executable model always gives a process schedule to explain its outputs[5]. After fully understanding the model, students can examine the system call traces and debug the libc source code with less pain.

Understanding synchronization. Synchronization primitives (mutexes, condition variables, semaphores, etc.) are usually informally introduced in a textbook or an operating system course. Implementing them upon our operating system model gives them a rigorous semantics specification[6]. Below displays a model of the buggy producer-consumer implementation from Chapter 30 of "The Three Easy Pieces" [3], in which a consumer may erroneously wake up another consumer (instead of a producer), resulting in a deadlock:

```
def Tworker(name, delta):
  for _ in range(N):
    while heap.mutex == '🔒':  # mutex_lock()
      sys_sched()              # |- spin wait
    heap.mutex = '🔒'          # |

    while not (0 <= heap.count + delta <= BUFSIZE):
      sys_sched()
      heap.mutex = '🔓'          # cond_wait()
      heap.cond.append(name)     # |
      while name in heap.cond:   # |- spin wait
        sys_sched()              # |
      while heap.mutex == '🔒':  # |- reacquire lock
        sys_sched()              # |
      heap.mutex = '🔒'          # |

    if heap.cond:                # cond_signal()
      t = sys_choose(heap.cond)  # |
      heap.cond.remove(t)        # |- wake up anyone
    sys_sched()

    heap.count += delta  # produce or consume

    heap.mutex = '🔓'  # mutex_unlock()
    sys_sched()

def main():
  heap.mutex = '🔓'  # 🔒 or 🔓
  heap.count = 0  # filled buffer
  heap.cond = []  # condition variable's wait list
  sys_spawn(Tworker, 'Tp', 1)  # delta=1, producer
  sys_spawn(Tworker, 'Tc1', -1)  # delta=-1, consumer
  sys_spawn(Tworker, 'Tc2', -1)  # delta=-1, consumer
```

At first glance, this model seems to diverge from the textbook example, as all synchronization primitives are denoted by spin-wait constructs (Lines 3-4, 11-12, and 13-14). However, this is intentional: spin wait reflects the specification that the thread could not make any progress unless the synchronization condition is satisfied (e.g., a mutex is in the unlocked state or a condition variable has been signaled). Blocking wait is merely one possible implementation. Such a model also captures a detail often overlooked by students: a condition variable contains an implicit re-acquisition of its associated mutex (Lines 13–15) after being signaled. An executable model facilitates the development of rigorous concepts in operating systems.

The incorrect use of condition variable is also non-trivial: manifesting the bug requires at least three threads (a producer and two consumers) and $N \geq 2$. Such a fact can be easily verified by the model checker (Section 4.1). Running this model under a uniform-random scheduler, there is only approximately an 8% chance of triggering the deadlock in which all three threads T_p, T_{c_1}, and T_{c_2} are spinning on Line 11.

Finally, we found that emojis in the code can improve the readability of program states: "🔒" intuitively indicates that a thread holds this mutex. Other cases include using 🙋 in Peterson's algorithm [34] (instead of `flag[2]` and integer values `0` or `1`) to indicate a thread "raising hand" to enter the critical section and 💥👍👮 to denote success or failure.

File system consistency and journaling. The emulated block device enabled us to implement ideas in file systems without tedious low-level device details. Recall that the block device is conceptually a `dict`. Thus, we can assign blocks with intuitive names like `'bitmap1'` to indicate a bitmap block in the persistent storage. We can also use this `dict` as a file system by mapping file names (e.g., `'/tmp/a.txt'`) to their metadata and contents (e.g., `('symlink', '/etc/passwd')`) when the actual storage layout is not relevant. Below is a simplified model of xv6 [10] log commit:

```
def main():
  # 1. log the write to block #B
```

[5]The model checker (Section 4.1) can be used to exhaustively examine all process schedules and understand the possible outputs.

[6]Our model assumes that the execution of statements between consecutive `sched()` appears to be atomic and uninterruptible.

```
  head = sys_bread(0)  # blocks #1, #2, ... are the log
  free = max(log.values(), default=0) + 1  # allocate log
  sys_bwrite(free, f'contents for #{B}')
  sys_sync()

  # 2. write updated log head
  head = head | {B: free}
  sys_bwrite(0, head)
  sys_sync()

  # 3. install transactions
  for k, v in head.items():
    content = sys_bread(v)
    sys_bwrite(k, content)
  sys_sync()

  # 4. clear log head
  sys_bwrite(0, {})
  sys_sync()
```

With the model checking feature (Section 4.1), all possible crash behaviors and potential file system inconsistencies can be exhaustively explored.

3.3 Application: Specification of Systems

An operating system model can be useful beyond explaining textbook cases. A model also provides a *behavioral specification* for real operating systems, like a high-level reference implementation. For example, it could be proved that the mutex model in Section 3.2 has the following two properties:

1. Safety: as long as a thread holds a mutex, any other thread's lock acquisition never returns.

2. Liveness: a thread eventually acquires a mutex if threads with acquired locks eventually release them under a fair (random) scheduler.

Because everything is a state machine (and thus a well-defined mathematical object), it could be theoretically possible to *prove* that a real system's implementation is consistent with a model by constructing a refinement mapping[7]. This is exactly the idea behind formally verified systems like seL4 [25] (with a Haskell executable model) and Hyperkernel [31] (with a Python executable model), which all modeled operating systems as a state machine. Even though the technical details of the research work may be too involved for first-time operating system learners, state machines still facilitate grasping the fundamental concepts underlying them–one could always perform a "brute-force prove" by enumerating all reachable vertices on the state transition graph for finite systems.

Models are also useful as a behavioral reference for real system implementations. A more practical "refinement mapping" is to feed the same workload to both a model and a real system. Cross-checking the model and system traces validates the implementation's correctness. For example, executing the same

[7] One fundamental result of program verification is that refinement mappings between high-level and low-level specifications always exist [2].

```
1 Q ← {[]};                          // the queue of traces pending checking
2 S ← ∅;                             // the set of checked states
3 while ¬Q.empty() do
4     tr ← Q.pop() ;
5     ⟨s, choices⟩ ← replay(tr);
6     if s ∉ S then
7         S ← S ∪ {s};               // add the unexplored state to S
8         for c ∈ choices do
9             Q.push(tr :: c);       // extend tr with c and append to Q
```

Algorithm 1: The MOSAIC model checker

`fork()` sequence (assuming that all forks succeed) should yield identical process trees for both the model and a student's operating system kernel. Such an approach is also known as the lightweight formal method [22] and has been widely adopted in validating practical systems [6].

4 One Model Checker to Rule Them All

> **Philosophy 3**: Enumeration demystifies operating systems.

The executable model's behavior can be exhaustively explored by enumerating all possible non-deterministic choices. This section presents such a model checker (Section 4.1) and its application to operating system teaching (Section 4.2), followed by short quantitative experiments in Section 4.3.

4.1 MOSAIC Model Checker Design and Implementation

Instead of executing a system call immediately, all MOSAIC systems calls return a `dict` mapping possible choices (which can be regarded as labeled transitions in the state machine) to lambda callbacks for actually performing the system call, even if there is only one unique choice:

```
def sys_sched(self):
  return {  # all possible choices
    f't{i+1}': (lambda i=i: self._switch_to(i)) # callback
      for i, th in enumerate(self._threads)
        if is_runnable(th.context)
  }

def sys_fork(self, *args):
  return {  # only one choice
    'fork': (lambda: self._do_fork())
  }
```

Such a design yields a simple replay-based state space explorer as shown in Algorithm 1. The algorithm is a straightforward breadth-first search that memorizes traversed states in S. A trace is a chronological list of each system call's selected choice. Replaying a trace will always reach the next system

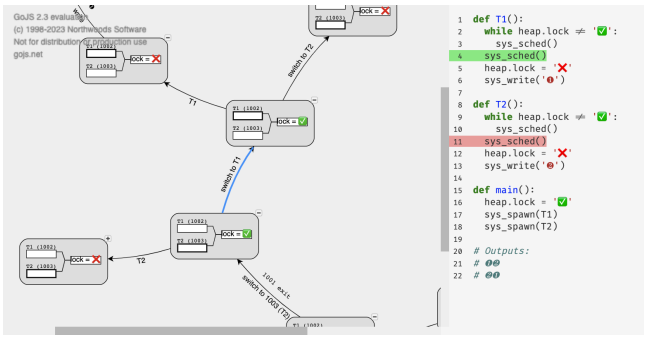


Figure 3: The interactive thread interleaving space explorer on MOSAIC's results of checking a spin lock implementation. Process and thread states are plotted as vertices. Thread program counters are highlighted on the source code like a debugger. Clicking a vertex expands its children.

call's non-deterministic choices (Line 5), or there is no choice ($choices = \varnothing$) when all processes and threads are terminated.

For finite-state models, the algorithm always terminates and produces a state transition graph whose vertices are traces in S and edges are labeled with c in Line 9. MOSAIC serializes the state transition graph as a JSON file. Both states (generator states, heaps, debug console output, and storage state) and transitions (labeled edges) are serialized. We encourage the students to follow the UNIX philosophy and pipe the text output to different backends:

1. Simply `grep stdout | sort | uniq -c` for a quick (and dirty, perhaps unsound) check for all possible debug console outputs.

2. Any JSON query or viewer like `jq` [15] to extract fields of interest (e.g., variable values or block device contents).

3. Our interactive state explorer (Figure 3) in which one can selectively expand nodes in the state transition graph. This interactive explorer is particularly handy for class demonstration.

4.2 Model Checking for Fun and Profits

The ability to exhaustively explore the state space makes a model checker suitable for rigorously explaining non-trivial cases in operating systems. A few such cases are shown below.

Processes and TOCTTOU attack. Both UNIX and our operating system model lack a mechanism (e.g., transactions [13, 37]) to enforce the atomicity across system calls and may be subject to time-of-check to time-of-use attacks. We demonstrate such a case of process-level race from [46]:

```
1 def main():
2   sys_bwrite('/etc/passwd', ('plain', 'secret...'))
3   sys_bwrite('file', ('plain', 'data...'))
4   pid = sys_fork()
5   sys_sched()
6   if pid == 0:  # attacker: symlink file -> /etc/passwd
7     sys_bwrite('file', ('symlink', '/etc/passwd'))
8   else:  # sendmail (root): write to plain file
9     filetype, contents = sys_bread('file')  # for check
10    if filetype == 'plain':
11      sys_sched()  # TOCTTOU interval
12      filetype, contents = sys_bread('file')  # for use
13      match filetype:
14        case 'symlink': filename = contents
15        case 'plain': filename = 'file'
16      sys_bwrite(filename, 'mail')
17      sys_write(f'{filename} written')
18    else:
19      sys_write('rejected')
```

MOSAIC reveals that "`/etc/passwd written`" is possible and gives such a process schedule. The exhaustive search can also reveal that the two `sys_sched` in Lines 6 and 10 are essential to produce such a result.

Hardness of shared-memory concurrency. Understanding thread interleaving can be difficult. Restoring the global ordering of shared memory accesses on thread-local read/write sequences is NP-Complete [17]. One interesting case is the possible outcomes of concurrent `tot++`, assuming that loads and stores are atomic (i.e., a sequentially consistent memory model) and the compiler does not merge multiple `tot++`:

```
def Tsum():
  for _ in range(N):
    tmp = heap.tot  # load(tot)
    sys_sched()
    heap.tot = tmp + 1  # store(tot)
    sys_sched()

def main():
  heap.tot = 0
  for _ in range(T):
    sys_spawn(Tsum)
```

MOSAIC reveals that `tot` can be 2 regardless of N and T (for $N, T \geq 2$) and gives such a thread schedule in which one thread "holds" a value of 2 in the last iteration of the loop and does not write it back until all other threads are terminated. We used the $N = 3, T = 2$ case as an exam problem, and approximately half of the students got wrong.

Persistence and crash consistency. Upon `crash()`, MOSAIC automatically explores all 2^n possible crash disks, assuming that any of the n buffered block I/O requests could be lost [36]. By modeling a file operation that involves multiple block updates (inode, bitmap, and data), an instructor can clearly and rigorously illustrate potential inconsistencies in a file system upon a system crash. Below is a textbook case in Chapter 42 of "The Three Easy Pieces" [3]:

```
def main():
  # intially, file has a single block #1
  sys_bwrite('file.inode', 'i [#1]')
  sys_bwrite('used', '#1')
  sys_bwrite('#1', '#1 (old)')
```

Subject	Parameters	# State	Memory	Time
fork-buf (7 LOC)	$n=1$ ($p=2$)	15	17.0 MB	<0.1s
	$n=2$ ($p=4$)	557	19.8 MB	3.3s (171 st/s)
	$n=3$ ($p=8$)			Timeout (>60s)
cond-var (34 LOC)	$n=1;t_p=1;t_c=1$	33	17.3 MB	<0.1s
	$n=1;t_p=1;t_c=2$	306	19.7 MB	0.1s (2 912 st/s)
	$n=2;t_p=1;t_c=2$	2 799	26.0 MB	0.8s (3 343 st/s)
	$n=2;t_p=2;t_c=1$	4 666	30.5 MB	1.4s (3 247 st/s)
xv6-log (27 LOC)	$n=2$	55	17.3 MB	<0.1s
	$n=4$	265	19.2 MB	<0.1s
	$n=8$	6 157	40.2 MB	1.3s (4 810 st/s)
	$n=10$	28 687	93.9 MB	20.7s (1 385 st/s)
tocttou (24 LOC)	$p=2$	33	17.4 MB	<0.1s
	$p=3$	97	17.8 MB	0.2s (413 st/s)
	$p=4$	367	19.4 MB	2.7s (135 st/s)
	$p=5$	1 402	23.5 MB	30.2s (46 st/s)
parallel-inc (11 LOC)	$n=1;t_s=2$	40	17.2 MB	<0.1s
	$n=2;t_s=2$	164	18.0 MB	<0.1s
	$n=2;t_s=3$	6 635	37.4 MB	1.4s (4 580 st/s)
	$n=3;t_s=3$	52 685	139.5 MB	14.1s (3 725 st/s)
fs-crash (25 LOC)	$n=2$	90	17.5 MB	<0.1s
	$n=4$	332	19.4 MB	<0.1s
	$n=8$	5 136	36.2 MB	2.6s (1 944 st/s)
	$n=10$			Timeout (>60s)

Table 2: Evaluation subjects and results. p, t, n denote the number of processes, threads, and loop iterations, respectively. All experiments were performed on an i7-6700 Linux PC with 4 GB RAM running Python 3.11. Each configuration is repeated for 10 times, and the average number is reported.

```
6   sys_sync()
7
8   # append a block #2 to the file
9   sys_bwrite('file.inode', 'i [#1 #2]')  # inode
10  sys_bwrite('used', '#1 #2')  # bitmap
11  sys_bwrite('#1', '#1 (new)')  # data block 1
12  sys_bwrite('#2', '#2 (new)')  # data block 2
13  sys_crash()  # system crash
14
15  # display file system state at crash recovery
16  inode = sys_bread('file.inode')
17  used = sys_bread('used')
18  sys_write(f'{inode:10}; used: {used:5} | ')
19  for i in [1, 2]:
20    if f'#{i}' in inode:
21      b = sys_bread(f'#{i}')
22      sys_write(f'{b} ')
```

MOSAIC's self-explanatory outputs verified that the one-page informal arguments in the textbook are indeed exhaustive and correctly covered all possible cases. MOSAIC can also check the journal implementation in Section 3.2 by adding `crash()` to the code and reveal that removing the `sync()` in Line 6 may result in file system inconsistency.

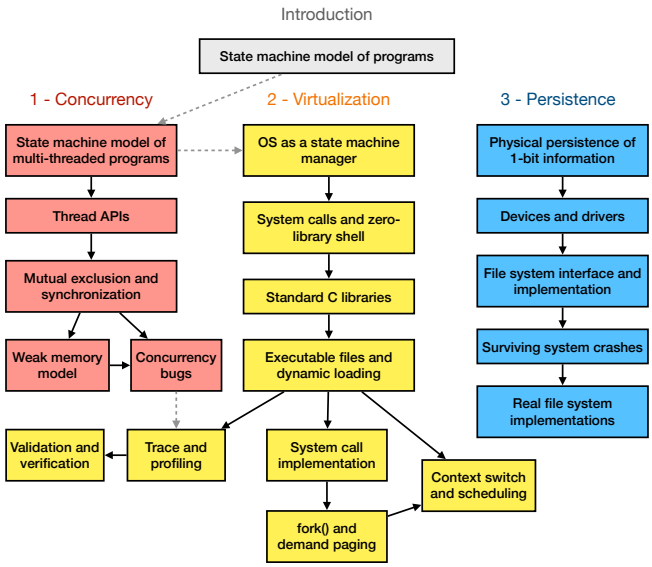


Figure 4: Major modules and their dependencies in our operating system course. The concept of state-machine is a good foundation for thread-based concurrency, and thus we introduce concurrency first in the course.

4.3 Experiments

We evaluate the performance of MOSAIC by checking the six representative models in Sections 3.2 and 4.2. Both experimental subjects and results are listed in Table 2. As expected, MOSAIC cannot address the state space explosion problem and has no comparable performance with a state-of-the-art software model checker with dedicated optimizations. Furthermore, programs that extensively fork is significantly slower (benchmarks fork-buf and tocttou) because our `fork()` is implemented by a full-system replay. Nevertheless, checking thousands of nodes per minute could be considered sufficiently useful for instructional purposes, and our design choice is to make a functional model checker minimal and elegant.

5 A New Operating System Course

We design a new operating system course from scratch based on "The Three Easy Pieces" [3] and our teaching philosophies: everything is a state machine, emulate state machines with executable models, and enumeration demystifies operating systems. The course syllabus is shown in Figure 4. This section presents the impacts of the state machine perspective (Section 5.1) and model checker (Section 5.2) in the course design, followed by discussions in Section 5.3.

5.1 State Machines and Operating Systems

In addition to introducing the key concepts in operating systems using state machines, the state machine perspective also

brings the following advantages in establishing a high-level understanding of important concepts regarding computer systems in a natural and coherent way.

Don't panic in hacking real systems! All students had a hard time in debugging real (even minimal) systems, including but not limited to operating system kernel, even if we provided skeletal code, tool chain, and state visualization scripts.

The state-machine perspective provides a natural reflex on how to deal with bugs or unexpected behavior in real systems: All bugs in computer systems are essentially some anomaly in the state-machine's execution trace. Given an unlimited amount of time, one just seeks the *first* abnormal state, and the root cause is right there. We teach students this (impractical) debugging principle and motivate students to consider clever tricks to make this procedure fast, robust, and easy.

For example, the essence of printf-debugging is to provide a high-level digest of the state-machine trace, which helps in narrowing down the scope of the initial anomalous state. One can also employ defensive programming by inserting assertions to the validity of states. These lessons are usually less taught in an operating system class but are essential for surviving hacking or implementing a large-scale system.

One classroom story is using a profiler (i.e., "frequent" state sampler) in diagnosing an unexpected 100% CPU usage on an idle workload in a production system in on a specific machine. The `perf` tool [9] attributes the hot spot to an `xhci`-related function, which leads us to a short-circuited USB port.

Concurrency meets state machines. The model checking community has long represented concurrent programs as state transition systems, and model checking is widely recognized as a computationally intensive technique that frequently encounters state explosion issues. Nevertheless, employing exhaustive enumeration is not the sole efficient approach to harness the capabilities of state machines.

The concept of data race, an important topic in operating system courses, refers to the simultaneous access of a shared memory location by two threads or processors (with at least one performing a write). Data races are considered harmful in systems programming.

When one checks a state machine trace against data races, it is essential to examine all types of state transitions that could lead to memory access [41]. However, two sources of memory access may be overlooked by students: (1) fetching an instruction from the program counter and (2) stack operations, including function and interrupt returns.

We let the students experience a subtle data race in an operating system kernel lab that requires students to migrate a process from one processor to another. The destination processor could not immediately schedule the process. Otherwise, there will be a data race on the kernel's interrupt stack.

Demystifying compilers. It is not obvious to students that C programs can also be represented by state transition systems. We use the example in Figure 5 (a non-recursive "Tower of Hanoi" implementation) to illustrate that the "runtime state" of C programs consists of static variables, heap memory, and a list of stack frames. State transitions are small-step expression evaluations at the top-most stack frame's program counter.

```c
void hanoi(int n, int from, int to, int via) {
  if (n == 1) {
    printf("%d -> %d\n", from, to);
  } else {
    hanoi(n - 1, from, via, to);
    hanoi(1, from, to, via);
    hanoi(n - 1, via, to, from);
  }
}
```

```c
typedef struct { int pc, n, from, to, via; } Frame;
#define call(...) ({*(++top) = (Frame) {0, __VA_ARGS__};})
#define ret()     ({top--;})
#define jmp(loc)  ({f->pc = (loc) - 1;})

void hanoi_nr(int n, int from, int to, int via) {
  Frame stk[64], *top = stk - 1, *f;
  call(n, from, to, via);
  while ((f = top) >= stk) {
    switch (f->pc) {
      case 0: if (f->n == 1) {
        printf("%d -> %d\n", f->from, f->to); jmp(4);
      } break;
      case 1: call(f->n - 1, f->from, f->via, f->to); break;
      case 2: call(1, f->from, f->to, f->via); break;
      case 3: call(f->n - 1, f->via, f->to, f->from); break;
      case 4: ret(); break;
      default: assert(0);
    }
    f->pc++;
  }
}
```

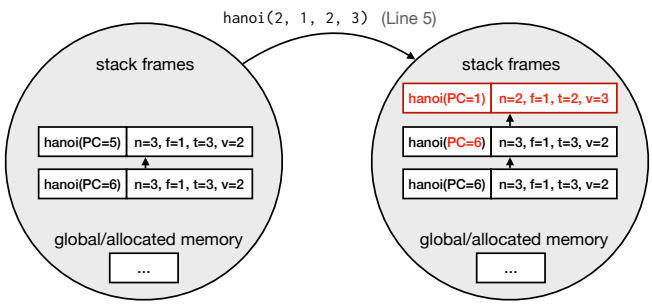


Figure 5: State machine perspective of C programs. `hanoi_nr` is also an "executable model" emulating recursions for rigorously understanding the semantics of C programs.

Compilers should always generate *equivalent* assembly (low-level state machine specification) from source code (high-level state machine specification). Therefore, a fundamental question is what kinds of translation are allowed for an optimized compiler. Notably, such deliberations are frequently neglected throughout the undergraduate curriculum, including in courses specifically addressing compilers.

With the conceptual model of state machines, the correctness of translation is essentially the equivalence between two state machines. This naturally leads to the definition of external observable equivalence between systems: given that system calls are the only way of influencing the remaining parts of the system, two programs are considered equivalent if they generate identical system call traces for the same inputs, and one program terminates if and only if the other program terminates. This principle serves as the core concept behind a verified compiler such as CompCert [27].

5.2 Modeling and Model Checking in Action

Models and emulation are everywhere. Models in an operating system course may not be limited to Python-implement system calls. We advocate using minimal but functionally "working" models, even they are implemented using a lower-level programming language.

One particular example is that we long had difficulties in explaining the ELF dynamic linker and loader to the students due to the unnecessarily excessive complexity of the ELF format. We identified that the problem stems from the fact that the ELF design is intended to be read exclusively by machines, rather than humans.

Therefore, we design a simplified binary format implemented using GNU C preprocessor and assembly. The binary file contains merely a magic number, a `NULL`-terminating symbol table whose entries are macros like `IMPORT(printf)` or `EXPORT(main)`, followed by assembly instructions. By reusing GCC and binary utilities, we implement the full toolchain of linker, loader, and an `objdump` equivalent in 200 lines of C code. Student and social media feedback indicate that such a model significantly flattens the learning curve of dynamic loading.

Formal method meets operating systems. We motivate the need for a model checker by making substantial (boring) efforts to draw a state transition graph to prove the safety and liveness of Peterson's mutex algorithm [34]. It is then obvious that a program like MOSAIC can replace human labor by emulation. We received positive feedback from students on their first contact with the model-checking approach, particularly the interactive visualizer (Figure 3), which is embedded in a Jupyter notebook for in-class demonstrations. The machine-generated state transition graph is also generally more reliable than the informal arguments in popular textbooks [42].

Another advantage of a model checker over existing teaching methods is the immediate feedback when answering "what if" questions related to changes in assumptions, implementations, and other factors. We encourage students to extensively experiment with the model, e.g., to see if the system breaks with added `sched()` or removed `sync()`.

The gap between models and real systems. We also teach students that models do not fully reflect the real world. Models are good at making *all* assumptions explicit, e.g., MOSAIC assumes the atomicity of statements between consecutive `sched()` calls and a sequentially consistent memory model.

The discrepancies between a model and an actual system are explained by careful examination of these assumptions. Peterson's algorithm is correct only under proper assumptions–specifically, a sequentially consistent memory model as if context switches only happen on instruction boundaries. For Peterson's algorithm, we provide an equivalent C implementation to illustrate how compiler and memory barriers may impact the program's behavior.

5.3 Student Acceptance and Discussions

Student Feedback. After publicizing the course lecture notes, demonstrations, and videos on the Internet, we received an excessive amount of positive feedback. Comments included statements like, "It is remarkable that such a comprehensive explanation of operating system principles can be provided in an undergraduate-level course." Students conveyed that they "gained valuable insights on overcoming the panic in hacking large-scale systems in this course."

There are also controversial arguments on the appropriateness of incorporating state machines as a key concept in operating system courses. However, we have also received feedback from the industry professionals supporting our approach by indicating that state machines are one of the most fundamental abstractions for controlling complexity in building production systems.

Since the first public release of the course in 2020, the video has received more than 2,000,000 plays on the Internet. Moreover, this course has been conferred the "Test-of-Time Teaching Award of the Department," as chosen by alumni who evaluated all courses in their curriculum.

Usefulness of models. Modeling is a versatile technique for establishing concepts and understanding. Modeling can also control the complexity by selectively hiding low-level implementation details. Another major advantage of executable models is making operating system concepts *rigorous*. Concepts (e.g., mutex, condition variable, and crash consistency) can be defined by "all possible behaviors on a model."

One may argue that any model behavior can be manifested by real workloads, and thus students should have first-hand experiences on real systems. We consider understanding the model (and thus the concepts) a critical step before students can hack real systems. Otherwise, the excessive and irrelevant implementation details can be a significant source of distraction.

Limitations. The "state-machine perspective" motivates the key insights and high-level designs of operating systems well. However, such over-simplification may yield students overlooking the challenges of implementing real systems. Therefore, we still consider the "hands-on approach," [26] in which students implement their own operating system kernel on

emulated bare metal, an indispensable part of an operating system course[8].

MOSAIC only models a small fraction of an operating system. More are missing, and one has to model them explicitly: file descriptors, signals, futexes, RAID, network stack, etc. Theoretically, it could be possible to model them in MOSAIC; however, we preferred simplicity in our model design and leaving these mechanisms to user-level applications like we did in Sections 3.2 and 4.2.

The implementation of MOSAIC also has limitations: `main` must be a Python generator (rather than a stackful coroutine). Thus, system calls are not allowed in functions being called by `main`. MOSAIC also assumes that the program being checked is deterministic. Non-determinism beyond system calls (e.g., random numbers) results in unsound model-checking results. Considering that MOSAIC is a pedagogical model checker and an instructor can easily bypass these limitations; thus, they are not a significant obstacle to adopting MOSAIC in practice.

6 Related Work

Emerging from the logic and programming language community, formal methods (mainly model checking and formal verification) has been widely adopted in the validation and verification of computer systems [6, 25, 27, 31, 47]. The key idea of formal methods is to treat specifications, models, and implementations as unambiguously-defined mathematical objects and *prove* properties by exhaustive search or axiomatic reasoning.

Despite a growing trend of formal method applications for computer systems, the teaching practice of "classical" operating systems remains classical on the layered abstractions of computer systems [3, 42] and the "hands-on" approach [26] in which students hack teaching operating system kernels [10, 19, 35] over emulators like QEMU [5] to fully understand all low-level implementation techniques.

There are attempts to incorporate model checking in teaching computer systems. Hamberg and Vaandrager [18] modeled textbook concurrency control algorithms using the Uppaal modeling language and checker. Michael et al. [28] target real Java programs on a message-passing model and check against all possibilities of message reorderings, drops, and duplications. Both concurrent programs and distributed systems are classical application scenarios of a model checker. To the best of our knowledge, we are the first to apply formal methods throughout an entire operating system course.

MOSAIC models a fully functional operating system by the unified treatment of non-determinism in system calls (Section 4.1). MOSAIC can check the interactions between processes, threads, and devices. Such a design resembles the EXPLODE system [47] for model checking real storage systems, in which all non-determinism and fault injection are implemented upon `choose()`.

As a pedagogical model checker, MOSAIC's primary use is to explain real operating system behaviors by mapping the model's execution traces (e.g., examples in Sections 3.2 and 4.2) to real systems. Such an approach belongs to the paradigm of lightweight formal methods [21, 22], which strongly emphasizes practicability rather than the full soundness of a proof. Lightweight formal methods have been proven effective against validating excessively complex real systems [6]. Like other pedagogical model checkers [28], we intentionally trade off the performance with understandability. Compared with fully verified systems [31], MOSAIC is functional but with magnitudes less code.

Emulation is also a widely-adopted approach in operating system teaching, which facilitates students establishing a correct and rigorous understanding of concepts. The exercises of "Three Easy Pieces" [3] are based on a substantial amount of independent emulators. MOSAIC as a unified model, on the other hand, can model (and check) the interplay between different levels of system mechanisms, e.g., how file system operations and process-level race result in a TOCTTOU attack in Section 4.2.

Finally, (replicated) state machines also play a fundamental role in distributed systems [32]. Formal methods became increasingly necessary in handling the counter-intuitive corner cases often overlooked by informal arguments. We believe that getting familiar early with such a paradigm on rigorous modeling and reasoning in a first operating system course can inspire the future generation of system researchers.

7 Conclusion

This paper presents a state-machine-first and model-based approach to teaching operating systems. By leveraging modeling and model checking, we can define operating system concepts rigorously, explore system behaviors exhaustively, and motivate non-trivial research systems intuitively under a unified framework in a first operating system course. We believe that this paper's teaching philosophies have the potential to lead a paradigm shift in the teaching of operating systems.

Acknowledgments

We would like to thank Haibo Chen, Yubin Xia, the anonymous reviewers, and our shepherd, David Cock, for their valuable and constructive feedback on this work. This work is supported in part by National Natural Science Foundation of China (Grant #61932021, #62025202, #62272218), Fundamental Research Funds for the Central Universities (#2022300287, #020214380102), State Key Laboratory for Novel Software Technology, and the Xiaomi Foundation.

[8] Students all had a hard time debugging a bare-metal kernel. Such experiences further motivate the need for debugging aids and dynamic analysis in Section 2.3.

References

[1] the musl libc. https://musl.libc.org/.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[4] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS 19, pages 14–22, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference*, USENIX ATC 05, Anaheim, CA, April 2005. USENIX Association.

[6] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP 21, pages 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI 08, pages 209–224, USA, 2008. USENIX Association.

[8] The Kernel Development Community. Linux tracing technologies. https://www.kernel.org/doc/html/latest/trace/index.html.

[9] The Kernel Development Community. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.

[10] Russ Cox, Frans Kaashoek, and Robert Morris. xv6: a simple, Unix-like teaching operating system. https://pdos.csail.mit.edu/6.1810/2022/xv6/book-riscv-rev3.pdf.

[11] 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 USENIX Symposium on Operating Systems Design and Implementation*, OSDI 2018, Carlsbad, CA, October 2018.

[12] The GDB Developers. GDB: The gnu project debugger. https://sourceware.org/gdb/.

[13] The Microsoft Windows Developers. The Windows Kernel Transaction Manager (KTM). https://learn.microsoft.com/en-us/windows/win32/ktm/kernel-transaction-manager-portal.

[14] Android Developer Documentation. Overview of memory management. https://developer.android.com/topic/performance/memory-overview.

[15] Stephen Dolan. jq: sed for JSON data. https://stedolan.github.io/jq/.

[16] 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, USA, 2002. USENIX Association.

[17] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[18] Roelof Hamberg and Frits Vaandrager. Using model checkers in an introductory course on operating systems. *SIGOPS Oper. Syst. Rev.*, 42(6):101–111, October 2008.

[19] David A. Holland, Ada T. Lim, and Margo I. Seltzer. A new instructional operating system. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE 02, pages 111–115, New York, NY, USA, 2002. Association for Computing Machinery.

[20] 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, pages 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2016.

[22] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *Computer*, 29(4):20–22, April 1996.

[23] 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, pages 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP 21, pages 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[25] 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, pages 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[26] Malcolm G. Lane. Teaching operating systems and machine architecture—more on the hands-on laboratory approach. In *Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE 81, pages 28–36, New York, NY, USA, 1981. Association for Computing Machinery.

[27] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[28] Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst, and Zachary Tatlock. Teaching rigorous distributed systems with efficient model checking. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys 19, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Ingo Molnar and Arjan van de Ven. Runtime locking correctness validator. https://www.kernel.org/doc/html/latest/locking/lockdep-design.html.

[30] 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 19, pages 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP 17, pages 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[32] 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, pages 305–320, USA, 2014. USENIX Association.

[33] Robert O'Callahan and Kyle Huey. rr: Lightweight recording and deterministic debugging. https://rr-project.org/.

[34] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[35] Ben Pfaff, Anthony Romano, and Godmar Back. The Pintos instructional operating system kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE 09, pages 453–457, New York, NY, USA, 2009. Association for Computing Machinery.

[36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI 14, pages 433–448, USA, 2014. USENIX Association.

[37] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP 09, pages 161–176, New York, NY, USA, 2009. Association for Computing Machinery.

[38] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan 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, pages 235–248, New York, NY, USA, 2005. Association for Computing Machinery.

[39] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. Debugging the OmniTable way. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 22, pages 357–373, Carlsbad, CA, July 2022. USENIX Association.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC 12, pages 28–37, USA, 2012. USENIX Association.

[41] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation*

and Applications, WBIA 09, pages 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[42] Abraham Silberschatz, Greg Gagne, and Peter B. Galvin. *Operating System Concepts*. Wiley, 10th edition, 2018.

[43] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. *ACM Transcation on Computer Systems*, 32(1), February 2014.

[44] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI 10, pages 33–46, USA, 2010. USENIX Association.

[45] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, first edition, 1987.

[46] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST 05, page 12, USA, 2005. USENIX Association.

[47] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 06, Seattle, WA, November 2006. USENIX Association.

[48] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys 10, pages 321–334, New York, NY, USA, 2010. Association for Computing Machinery.

Accelerating Distributed MoE Training and Inference with Lina

Jiamin Li
City University of Hong Kong

Yimin Jiang
ByteDance Inc.

Yibo Zhu
Unaffiliated

Cong Wang
City University of Hong Kong

Hong Xu
The Chinese University of Hong Kong

Abstract

Scaling model parameters improves model quality at the price of high computation overhead. Sparsely activated models, usually in the form of Mixture of Experts (MoE) architecture, have sub-linear scaling of computation cost with model size, thus providing opportunities to train and serve a larger model at lower cost than their dense counterparts. However, distributed MoE training and inference is inefficient, mainly due to the interleaved all-to-all communication during model computation.

This paper makes two main contributions. First, we systematically analyze all-to-all overhead in distributed MoE and present the main causes for it to be the bottleneck in training and inference, respectively. Second, we design and build Lina to address the all-to-all bottleneck head-on. Lina opportunistically prioritizes all-to-all over the concurrent allreduce whenever feasible using tensor partitioning, so all-to-all and training step time is improved. Lina further exploits the inherent pattern of expert selection to dynamically schedule resources during inference, so that the transfer size and bandwidth of all-to-all across devices are balanced amid the highly skewed expert popularity in practice. Experiments on an A100 GPU testbed show that Lina reduces the training step time by up to 1.73x and reduces the 95%ile inference time by an average of 1.63x over the state-of-the-art systems.

1 Introduction

Recent advances in deep learning have shown that a model's quality typically improves with more parameters [15, 21, 23, 29, 47]. Many new frontiers in Computer Vision (CV) and Natural Language Processing (NLP) have been explored using large dense models [22, 38, 44]. While effective in terms of model quality, the computation cost of model training and serving is extremely high. ChatGPT [1], an impressive chatbot released by OpenAI, is estimated to spend 3 million dollars per month to serve user requests. Wider adoption and development of these models are impeded by the exorbitant compute cost.

Following the basic idea of curbing the computation cost of massive models, *sparsely activated* models have recently been introduced [13, 23, 33, 44]. The *Mixture-of-Experts* (MoE) structure is now one of the most popular ways to implement sparse activation [13, 14, 44, 55]. For each input, instead of using all parameters, an MoE model selects just a few of them, i.e. *experts*, for processing. This leads to sub-linear scaling of FLOPs needed with model size. Recent literature [9, 22, 26, 30, 38, 54] has proven the potential of MoE models. For instance, Google develops a family of language models named GLaM using MoE [22]. Compared to GPT-3 with 175 billion parameters, the largest GLaM has 1.2 trillion parameters while only consuming 1/3 of the energy for training. Meanwhile, GLaM still achieves better zero-shot and one-shot performance than GPT-3. Microsoft reports that their MoE-based language models achieve a 5x training cost reduction compared to a dense model with the same model quality [38].

Given the uptake of MoE, there have been several systems for efficient MoE training and inference, including Google's Mesh TensorFlow [43], Meta's FairScale [10], Microsoft's DeepSpeed [2] and Tutel [7], etc. They provide APIs for users to replace the conventional dense layers with MoE layers with minimal code changes. They adopt both data parallelism and expert parallelism to accelerate the training and inference. That is, each device (e.g. GPU) is assigned with a unique expert, and uses all-to-all to receive inputs from other devices and then sends the gradients back to them accordingly. During training, allreduce is then used to aggregate non-expert gradients in the backward pass.

We focus on the efficiency of distributed MoE training and inference in this work. As some [25, 31, 41] has shown, the all-to-all operation is the main bottleneck. All-to-all blocks the subsequent computation operations and needs to be invoked two times in the forward pass and another two in the backward pass for each MoE layer. Interestingly, the main causes for all-to-all being the bottleneck are different in training and inference. In training, all-to-all and allreduce often contend for network bandwidth when they overlap in the backward pass,

leading to a prolonged blocking period to the computation. Inference, on the other hand, presents a highly-skewed expert popularity driven by real-world requests. Devices with popular experts have to handle much more data than others. Not only does it delay the launch of all-to-all, but it also causes imbalanced transfer size and bandwidth utilization across the devices, both of which are detrimental.

We are thus motivated to systematically tackle the all-to-all bottleneck. Our solution is Lina, a system that accelerates both MoE training and inference.

In training, we prioritize all-to-all over allreduce in order to improve its bandwidth. Existing MoE systems launch separate CUDA streams for the expert-parallel and data-parallel process groups which correspond to all-to-all for expert and allreduce for non-expert parameters, respectively. As there is no coordination between these streams, all-to-all and allreduce can overlap and fair-share the network bandwidth. Unlike allreduce, all-to-all is blocking and cannot be made parallel with the computation process. Thus, prioritizing all-to-all in the backward pass and avoid concurrent allreduce is crucial to reducing the blocking period.

To efficiently prioritize all-to-all, we adopt tensor partitioning which breaks down a tensor into smaller chunks, each of which forms a micro-op. With micro-ops, simple priority scheduling can be applied to guarantee full bandwidth for all-to-all while allowing allreduce micro-ops to make progress when all-to-all is not present. In addition, micro-ops allow the expert computation to be pipelined with all-to-all.

In inference, we dynamically schedule the resources for each expert in order to balance the workload of each device, thereby alleviating the imbalanced all-to-all transfer size and bandwidth. Intuitively popular experts should be given more resources while the rest may be served with less resources. The key challenge here is to efficiently and accurately obtain the expert popularity *before* the selection is actually done by the gating network, for every batch of input at each MoE layer, so scheduling benefit can be maximized with minimal overheads. Fortunately, we find the experts selected by each token across the layers demonstrate clear patterns, which allow us to estimate the expert distribution of the upcoming layer based on the past selection results from the preceding layers. We adopt a two-phase scheduling approach that fine-tunes the estimation based allocation only when the actual expert popularity deviates too far.

We build Lina based on DeepSpeed MoE [2] and PyTorch, and evaluate it on a cluster with up to 16 Ampere A100 GPUs with 40GB memory and 100Gbps InfiniBand. Results show that Lina accelerates all-to-all by at least 2.21x, and achieves on average 1.57x speedup in overall training step time compared to state-of-the-art system DeepSpeed. The median and 95%ile inference time is reduced by 1.45x and 1.63x.

Our contributions can be summarized as follows:

- We present an in-depth empirical analysis of distributed MoE to show the main causes for all-to-all to be the performance bottleneck in training and inference.
- We propose to prioritize all-to-all over allreduce in order to improve its bandwidth and reduce its blocking period in distributed training. Lina's scheduler incorporates tensor partitioning and pipelining to perform micro-op scheduling.
- We examine the pattern in expert selection of MoE layer and propose to estimate the expert popularity to conduct resource scheduling in advance during inference. Lina adopts a two-phase scheduling scheme to minimize the overhead.
- We implement a concrete prototype system and conduct comprehensive testbed experiments to demonstrate the benefits of our design in a realistic GPU cluster setting.

2 Background and Motivation

We start with an introduction on MoE and a widely-adopted distributed system for MoE model in §2.1. Then, we motivate our idea by analyzing the performance bottleneck (i.e. all-to-all) in distributed MoE training and inference in §2.2.

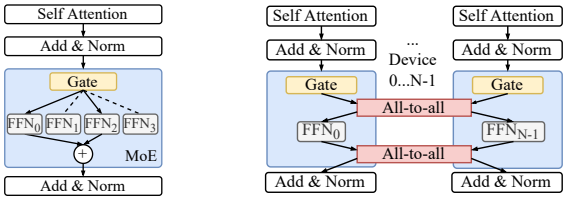
2.1 A Primer on MoE

Mixture-of-Experts (MoE) has been adapted to different types of DNN models , and exhibits great potential in improving the performance of language models in particular. GShard [31] and Switch Transformer [23] are two seminal works on scaling Transformer-based language models with MoE layers. We focus on MoE in Transformer-based models in this work.

Transformer-based models normally use an MoE layer to replace the feed-forward network (FFN) layer. An MoE layer consists of multiple FFNs each serving as an *expert*, and a gating network (Figure 1a). Every expert is a fully-connected two-layer network using ReLU activation but with different parameters. The gating network takes in the embedding vector of each token and multiplies them with its trainable matrix. Based on the results, it dispatches the token to a small number of experts (usually one or two). The final output of the MoE layer is the weighted sum of outputs from the selected expert(s). The sparsity nature of MoE improves the model scaling in size without increasing the training cost and naturally leads to a dynamically-changing model graph.

Load balancing loss. In MoE training, an auxiliary loss is introduced to evaluate the token distribution among the experts [23]. The objective is to achieve a uniform distribution of tokens across the experts, thereby preventing an excessive concentration of tokens in a single expert. By minimizing this loss term, we encourage but do not enforce the gating network to produce a perfectly balanced token distribution.

The standard practice is to calculate the auxiliary loss of each MoE layer and sum them with the training loss using an appropriate weight. Previous research has demonstrated the effectiveness of this approach [16,33,45]. However, it should be noted that achieving a perfectly balanced distribution, where the auxiliary loss converges to zero, is challenging [18, 53].



(a) There are four experts and the gate selects two experts.

(b) Distributed MoE. Data parallelism and expert parallelism are used.

Figure 1: MoE layer in Transformer-based models.

# Experts	Model	Training (ms)		Inference (ms)	
GPUs	#Layers & Params	All-to-all	Ratio	All-to-all	Ratio
4	12L + 117M	259	36.7%	73	27.4%
	24L + 233M	589	35.4%	103	26.2%
	36L + 349M	979	38.2%	153	28.3%
16	12L + 419M	333	39.5%	102	32.5%
	24L + 838M	715	37.6%	177	31.7%
	36L + 1.2B	1145	36.8%	243	27.4%

Table 1: The completion time of all-to-all and its ratio in training and inference task of Transformer-XL [20] in different number of experts per layer. Training and inference have the same batch size here. Each FFN layer is replaced with MoE and the number of experts is equal to the number of GPUs similar to the common practice [23]. A100 GPUs with 40GB memory and 100Gb/s InfiniBand are used. We use the MoE implementation in DeepSpeed.

During MoE inference, the trained gating network is utilized to dispatch tokens to the experts based on their respective embeddings. This process is solely driven by the characteristics of the token embeddings.

Hybrid parallelism in distributed MoE. Training and serving MoE models in a distributed manner are necessary due to the tremendous compute requirement of large-scale language models [12]. For efficiency, both data parallelism and MoE-specific *expert parallelism* (as a form of model parallelism) are applied [23,31]. Existing MoE systems [2,7,10,23,31,43] allocate one unique compute device (e.g., GPU) for each expert in expert parallelism. An all-to-all communication is then needed to send tokens to their experts selected by the gating network, and another all-to-all is needed to send tokens back to the device they belong to in data parallelism to finish the rest of the forward pass as shown in Figure 1b.

2.2 Bottleneck Analysis

Much prior work has identified that the introduction of all-to-all in MoE causes performance inefficiency in Transformer-based models [7, 41, 54]. We extract the completion time of all-to-all operations in both training and inference in our GPU cluster as shown in Table 1. All our experiments in this section use the same testbed and settings. Overall, all-to-all takes an average of 34.1% — a significant fraction of the step time. Interestingly, though the bottleneck brought by all-to-all is universal in both MoE training and inference, the causes differ. In the following, we motivate our work by analyzing how all-to-all affects the efficiency of training and inference, respectively.

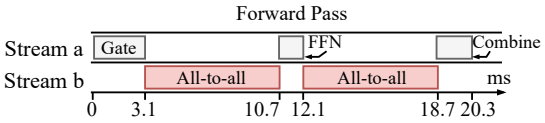


Figure 2: Timeline of forward pass an MoE layer. We simplify the presentation by bundling GPU kernels here: The computation kernels are grouped by their roles in the MoE layer into Gate, FFN and Combine. The Combine operation involves reshaping the tensors and computing the weighted output. The timeline is taken from a sample run of the 419M-parameter model in Table 1.

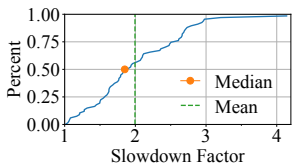


Figure 3: CDF of how much all-to-all is prolonged when it overlaps with allreduce operation. We mark the median and average slowdown factors.

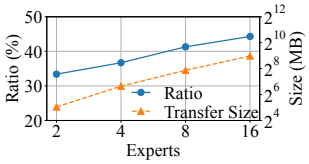


Figure 4: The proportion of all-to-all's completion time over training step time when the number of experts grows. Dashed line plots the data size in one all-to-all operation.

Synchronous all-to-all with large data transfer. The common characteristic shared by MoE training and inference is all-to-all's large data transfer. All-to-all is an irreplaceable synchronous component to handle the data exchange among devices in MoE layer. Each MoE layer has two all-to-all operations to send the tokens to the experts and then restore the position of tokens, as introduced in §2.1. The data transfers in the two all-to-all operations have the same size because the expert's FFN architecture ensures that its input data size is the same as the output data size. Figure 2 shows an empirical timeline view of the forward pass of MoE model in our cluster. All-to-all takes 74.9% of the end-to-end running time of one MoE layer. Expert FFN computation and the combine operation follow when all-to-all operation completes. MoE training and inference suffer from such inefficiency consistently. GPU is mostly idle during this period: We use the PyTorch Profiler [6] to profile the GPU activities for 20 steps in each experiment in Table 1, and find that the average GPU SM efficiency during all-to-all is 3.7%. Besides, the data transfer size grows linearly with the number of experts. Figure 4 presents the empirical evidence of all-to-all's transfer size as the number of experts grows from 2 to 16 (128). With the increasing number of experts, the time taken by all-to-all grows from 33.4% to 44.5% of the step time.

Problem in training: Prolonged all-to-all with allreduce. The unique challenge in MoE training is that applying the hybrid parallelism creates a particularly severe impact to all-to-all in backward pass. Non-MoE layers in *data parallelism* need *allreduce* to aggregate the gradients, while *expert parallelism* requires all-to-all to exchange tokens to compute expert gradients. Since the two operations control their own process groups independently, two dedicated CUDA streams are launched concurrently. This is demonstrated in Figure 5 with the timeline of backward pass in a sample run of MoE training. As the two operations overlap, they contend for the network bandwidth and their completion times are severely

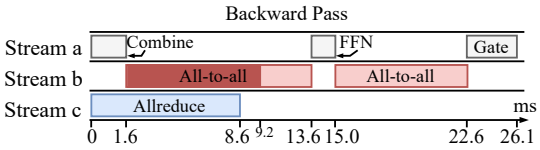
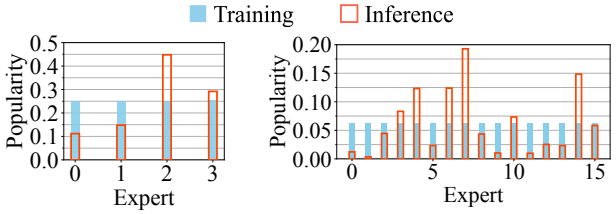


Figure 5: Timeline of backward propagating an MoE layer under hybrid parallelism. The first all-to-all is prolonged by the allreduce operation in Stream b. The shadowed part is its original completion time.



(a) 4-expert MoE. **(b)** 16-expert MoE.

Figure 6: Sampled expert popularity. The distribution is computed as the ratio between the number of tokens received by the expert and total number of tokens in one batch. We use the Enwik8 test set [3] for evaluation.

prolonged. To make matters worse, we find that the slowdown factor varies significantly. We collect the completion times of 1,500 all-to-all operations in backward pass on our testbed and plot the CDF of the slowdown factor they endure with allreduce in Figure 3. Observe that the median slowdown is over 1.83x and the worst is 4.14x.

Problem in inference: Skewed expert popularity. The main cause of all-to-all being the bottleneck in MoE inference is the skewed expert popularity. The token-to-expert distribution in inference is purely workload-driven, and we empirically find that the expert popularity is highly skewed in sharp contrast to training. We sample the expert popularity of the same MoE model in training and inference in Figure 6. In training, the distribution is nearly the same across all experts after hundreds of steps due to the use of load balancing loss. In inference, however, the most popular expert receives 4.02x and 5.56x tokens of the least popular ones in 4-expert and 16-expert inference tasks. With the same network and computation capacity, devices hosting popular experts take much longer to perform expert computation. In this experiment, the maximum idle time of the least popular expert is 29.4% of the inference time of that batch. Thus, within one batch, tokens to the less occupied experts have to wait for others to complete on the more popular experts, degrading the all-to-all performance significantly. Further, under uniform expert-device allocation, devices hosting popular experts have more tokens using their links for all-to-all, while the links of other devices are underutilized.

3 Design Overview

Lina is designed to accelerate all-to-all in distributed MoE. It attacks both the bandwidth contention with allreduce in training, as well as the straggler with unbalanced all-to-all bandwidth in inference. We focus specifically on MoE implementations that leverage both data and expert parallelism.

MoE training. We aim to improve the *bandwidth* of all-to-all in order to reduce the blocking period of the computation operations. Our key idea here is to *prioritize all-to-all* so it does not fair-share bandwidth with concurrent allreduce (§4). This is achieved using tensor-partitioning. We partition all-to-all and allreduce tensors into small chunks, each of which then forms a *micro-op*. Lina schedules an allreduce micro-op only when there is no all-to-all waiting or ongoing so that all-to-all is guaranteed the full network bandwidth during its lifetime. Without prior information, tensor-partitioning and micro-ops can ensure that in most cases all-to-all can launch immediately and allreduce is not deferred excessively.

MoE inference. We propose to dynamically adjust the device allocation for experts based on the *expert popularity*, so that is not all-to-all is not delayed by the trailing tokens, and its bandwidth utilization across links is balanced (§5). We exploit the expert selection pattern across adjacent layers to estimate the expert popularity. Based upon the estimation, Lina performs scheduling at each layer to allocates proportionally more devices for popular experts and pack unpopular ones to fewer devices, and coordinate all-to-all correspondingly.

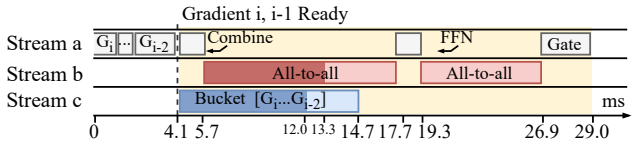
4 Prioritizing All-to-All Training

We have shown that all-to-all is slowed down significantly if it overlaps with allreduce in the backward pass in MoE training. Lina partitions the communication operations into small micro-ops and schedule them strategically in order to prioritize all-to-all without impeding allreduce and the computation process. We introduce the design challenges in §4.1. In §4.2, we present Lina's communication scheduler that uses tensor partitioning and pipelining to improve the training efficiency.

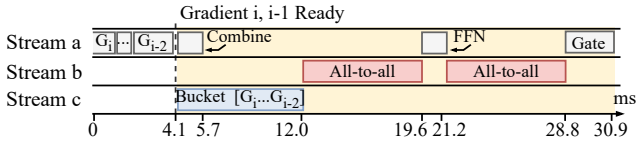
4.1 Design Challenge

Intuitively, Lina can prioritize all-to-all and avoid concurrent execution with allreduce with strict priority scheduling. All-to-all is always dispatched first if both are present in the queue, and subsequent operations have to wait until the running one finish to make sure allreduce does not share the bandwidth.

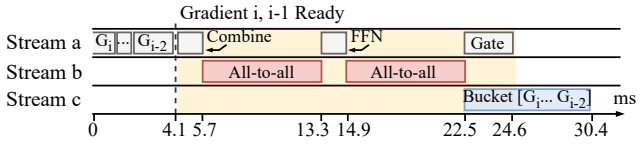
It turns out that simply prioritizing all-to-all is not as efficient as one may expect. For work-conservation, when an allreduce arrives first, it should be launched immediately. The problem is when an all-to-all arrives later, though ideally one would preempt the allreduce due to priority scheduling, this is not possible in current multi-GPU communication libraries such as NCCL [4]. The communication primitives are highly optimized and upon being called, their complete transmission strategies are settled and pushed to the CUDA streams. There is no control knob inside each primitive to adjust how it shares resources (e.g. CUDA cores, network bandwidth) with others. Thus, as the example in Figure 7b shows (based on testbed



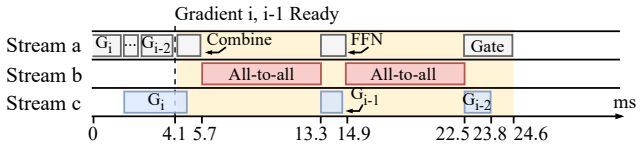
(a) Baseline. Shadowed all-to-all and allreduce are their completion times without concurrent operations. Computing the entire MoE layer's gradients ends at 29.0ms.



(b) Naively prioritizing all-to-all without concurrent transmission can lead to worse results; computing the MoE layer's gradients ends at 30.9ms. The completion time is profiled. Theoretically, the completion time should be the same as Figure 7a.



(c) Deferring allreduce to after the second all-to-all leads to better training efficiency; computing the MoE layer's gradients ends at 24.6ms.

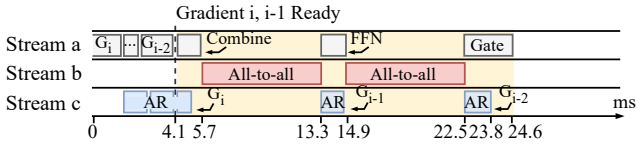


(d) Scheduling results if the arrival time and running time of communication operations are known a priori. The allreduce completes much faster than (c).

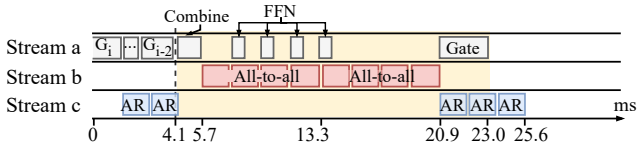
Figure 7: Backward pass of MoE training. The yellow background is the period of computing the gradients of the MoE layer. Stream a is responsible for the computation process and streams b and c are for communication. This timeline is extracted from a real run of the 419M-parameter benchmark model in Table 1.

experiments), naively prioritizing all-to-all actually leads to a longer completion time for the first all-to-all and training step time compared to the baseline in Figure 7a.

A potential solution is to obtain the arrival time and running time of the upcoming all-to-all and allreduce, and orchestrate them accordingly to maximize the efficiency. Assuming we know that the allreduce for gradient i can complete before all-to-all and the completion time of gradient $i-1$'s allreduce is shorter than FFN computation. Then we can schedule gradient $i-1$'s allreduce to the gap between the two all-to-all operations at 13.3ms as depicted in Figure 7d. Obtaining the precise knowledge of arrival and running times is, however, a daunting task. ML frameworks such as PyTorch fuse gradients into buckets based on a user-defined bucket size to optimize allreduce efficiency. Yet in large Transformer-based models, gradient sizes are also large; since bucketing is done on the gradient boundary, the actual bucket size for allreduce varies wildly [5]. Moreover, the implementation details of allreduce make it difficult to acquire a reliable running time estimate as prior work has found out [19].



(a) Prioritize all-to-all and partition allreduce tensors. Instead of bucketing gradients, we partition gradient i into three chunks when it is computed.



(b) Tensor partitioning for all-to-all and pipeline the FFN computation.

Figure 8: We show the scheduling results from Figure 7a with tensor partitioning. All-to-all and allreduce micro-ops are of the same size.

The other design choice is to blindly defer allreduce until an even number of all-to-all finish as there should be a larger gap between the backward pass of two MoE layers relative to FFN's backward computation. Figure 7c shows the best scheduling result based on the baseline in Figure 7a. In this case, allreduce can be launched when the second all-to-all finishes and completes before the first all-to-all of the next MoE layer (not shown in the figure). Yet, in other (worse) cases, allreduce may still block the all-to-all of the upcoming MoE layer if it takes relatively longer. In the extreme case, no allreduce can be launched until all four all-to-all operations of the current step finish. Since devices have to wait for allreduce before moving onto the optimization phase, this incurs more delay and is undesirable for wait-free backward pass [51].

4.2 Tensor Partitioning and Micro-Ops

To resolve the above challenges, we propose tensor partitioning that breaks down a communication operation into micro-ops, which can be easily prioritized with high efficiency.

Tensor partitioning. Unlike tensor bucketing which fuses multiple gradients for an allreduce, Lina partitions each gradient tensor into equal-sized small chunks and executes individual allreduce *micro-ops* independently. This brings two advantages. First, it resolves the varying bucket size problem for allreduce since each micro-op is uniform in size now. Second, micro-ops naturally make better use of bandwidth [36] without causing too much delay to allreduce under priority scheduling. Consider the same setup from Figure 7a, in Figure 8a we partition gradients into five chunks. Before the first all-to-all arrives, Lina launches three allreduce micro-ops; after the first all-to-all ends, it starts another micro-op to opportunistically make use of the expert computation time. Compared to the scheduling result without micro-ops in Figure 7c, allreduce for gradient $i-2$ now completes 6.6ms or 21.7% faster without prolonging all-to-all. Tensor partitioning does incur overhead due to the partition and concatenation operations before and after an allreduce, but it is mild: the

overall overhead in Figure 8a's case is 764us. §7.2.2 has more details of the overhead analysis.

Pipelining micro-ops. Intuitively, we can also partition all-to-all which provides an opportunity to pipeline the expert FFN and further reduce the time that computation is blocked. Specifically, we can pipeline the expert computation and all-to-all micro-ops (Figure 8b). Since the FFN computation is in token granularity, the expert can start computing with a subset of the tokens after one all-to-all micro-op. With pipelining, we can eliminate the FFN time which is 1.6ms in this example.

Expert packing. Ideally, the FFN and all-to-all micro-ops should take a similar time so that both compute capacity and network bandwidth are fully utilized without any bubbles in the pipeline. However, we notice that a single FFN micro-op takes much less time than its corresponding all-to-all micro-op (Figure 8b). In Lina, we consider packing multiple experts on each device whenever possible to maximize the pipelining efficiency. Lina adopts the following approach: starting with one expert per device, it iteratively increases the number of experts per device in powers of two, until the FFN computation exceeds that of the all-to-all micro-op. In case of GPU memory shortage, we adopt DRAM-offloading [42] to transfer expert parameters that are not currently in use to host memory.

5 Scheduling Resources in Inference

Recall in §2.2, we have shown empirically that skewed expert popularity leads to unbalanced processing times across tokens of the same batch in MoE inference, which delays all-to-all and causes imbalanced bandwidth for it severely. The root cause lies in the data granularity mismatch between the expert and the attention layers in the model: an expert processes individual tokens, but the attention layer processes an entire sequence as a whole. Our design question is thus: How can we ensure that each token within the same batch experiences the same end-to-end completion time no matter its expert selection result? We will first discuss the challenge of achieving this through dynamic resource scheduling (§5.1), and then present our design that exploits the unique token-level expert selection pattern to address the challenge in §5.2.

5.1 Design Challenge

To cope with skewed expert popularity, intuitively one must accordingly adjust the resource allocation for experts. This adjustment also needs to be done for each input sequence as the expert popularity distribution varies across sequences. An immediate question is: how can we know the expert popularity distribution, before the input is processed by the gating network?

This question is challenging for two reasons. First, even for a given batch of input, expert popularity varies across MoE layers of the model. We collect the expert popularity of different MoE layers for 1000 batches of input requests. Table 2 shows the top-4 popular experts of two 12-expert inference tasks: text generation and translation. Observe that each MoE layer of the same task (model) has completely different popular experts. This also suggests that dynamic resource scheduling has to be done before each MoE layer in order to be effective. Moreover, scheduling resources according to the actual expert selection results, as some might be thinking, incurs delay in collecting information, making scheduling decisions, and coordinating the all-to-all amongst all experts with respect to the new expert-device mapping, all of which are blocking operations and are performed at each layer. This is far from optimal (as will be shown in §7.3.1). Thus, we need to know as much as possible the expert popularity *before* the gating network selects experts in each layer, so these overheads can be largely overlapped with MoE computation.

5.2 Popularity based Scheduling

Lina tackles the design challenge by exploiting the token-level expert selection pattern which we empirically establish now. Building upon this, we design a resource scheduler that replicates popular experts on proportionally more devices in order to better balance the workload.

Pattern in expert selection. Experts in MoE models are trained to specialize in different types of input. We find that a token's expert selection demonstrates a pattern across the MoE layers. Tokens that have selected the same expert in layer i tend to select the same expert again in layer $i+1$. We trace the expert selection of sampled tokens. For each group of tokens that have selected the same expert in layer i, we calculate the ratio of them that in the next layer also select one of the same top-k experts ranked locally among the same group. Figure 9 plots this ratio averaged over token groups in two 12-layer MoE models. We see 41.94% tokens exhibit this pattern when k is 1 and 54.59% when k is 2, and deeper layers see more tokens with this pattern.

This observation makes intuition sense. The gating network has a simple architecture, and their routing or expert selection decision is made (largely) based on relatively simple features, such as the parts of speech of a word (noun, verb, etc.), and the meaning of the word (number, time, etc.) [32]. These features are fixed for each token. Meanwhile, experts focus on the local syntax information of each token rather than the cross-dependency within a sequence. For all these reasons, similar tokens naturally tend to be processed by the same or similar experts in each layer.

Estimating expert popularity. Though this pattern may not be sufficient to predict a particular token's expert selection accurately, it provides enough clues for us to estimate the overall expert popularity for a given batch. Specifically, Lina's estimation approach works as follows. In the profiling stage, we collect the expert selection results of all tokens when the load balancing loss is minimized and becomes stable. We then

Model& Dataset	Layer	Top-4			
Transformer-XL & Enwik8 (Text generation)	3	9	4	5	10
	4	5	7	8	10
	8	9	2	3	13
	12	4	5	15	8
BERT-Large & WMT En-De (Translation)	6	7	6	10	1
	8	10	6	2	15
	10	9	4	11	8
	12	1	8	10	14

Table 2: Top-4 popular experts in sampled MoE layer of two MoE models.

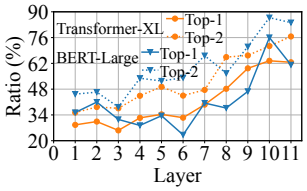


Figure 9: Ratio of tokens that select one of the top-k experts in layer $i+1$ given that they have selected the same expert in layer i.

group tokens that select the same experts from layer $i-l$ to layer i, which represent a unique sample path of experts used. For each sample path j, we compute the expert popularity distribution Ψ_j^{i+1} for layer $i+1$. Here l is the path length to control the accuracy-cost tradeoff in profiling: a larger path length leads to more accurate estimation for layer $i+1$ at the expense of higher data collection and computation costs.

Then based on the profiled distributions $\{\Psi\}$, Lina can estimate the next layer's expert selection distribution for each sample path of experts traversed by a token in inference (starting from the l-th layer of the model). In each layer i, for a sample path j, we pick the top-k expert(s) of the subsequent layer from Ψ_j^{i+1} and use their probabilities $\{P_j^{i+1}(e)\}$ to represent expert popularity for resource scheduling, where e denotes an expert. The reason why we only consider top-k experts is that they demand the most resources, and the remaining experts have low popularity (Figure 9). Note that this estimation happens before any MoE layer computation takes place.

Two-phase scheduling. During inference, Lina dynamically conducts layer-wise resource scheduling in two phases.

The first phase happens right after the expert popularity estimation at each MoE layer, when Lina relies on the estimation to replicate popular experts on more devices and pack unpopular ones onto fewer devices. Specifically, the total number of devices for expert e is determined by:

$$n_e = N \times \sum_{t=1}^{N_t} P_{j(t)}^{i+1}(e)/N_t, \tag{1}$$

That is, for the current batch of input with N_t tokens, using estimation from each token t's sample path $j(t)$ up to layer i, the overall popularity of expert e is estimated as $\sum_{t=1}^{N_t} P_{j(t)}^{i+1}(e)/N_t$ for layer $i+1$ accounting for all tokens. This requires the same proportion of devices assuming the expert parallelism degree is 1 (i.e. the number of devices equals the number of experts). For experts with the estimation n_e, we adopt the first-fit-decreasing heuristic to pack them into the empty devices so the total devices used are minimized. It is possible that some experts, being extremely unpopular (for this batch), are not amongst the top-k list of any tokens and thus do not have their n_e estimation. They are assigned evenly to the remaining free devices if any; otherwise are randomly assigned to a device.

In phase two, Lina fine-tunes the estimation-based scheduling decision after the gating network selects the actual experts. It checks if the selection result deviates significantly from the estimation, by comparing the overall top-$2k$ experts. If the two lists are identical, no fine-tuning is needed and inference continues. Otherwise, the scheduler re-computes the resource allocation with the actual expert popularity now available following the same logic in phase 1. The fine-tuning phase does incur delay to collect the gating outputs and check against the estimation, which is necessary to deal with inaccurate estimation that turns out to be much more detrimental to performance, if left unchecked (§7.3).

6 Implementation

We implement Lina on DeepSpeed MoE and PyTorch using C++ and Python. PyTorch 1.10, CUDA 11, and NCCL 2.10 are used. We modify PyTorch's implementation of distributed training to support Lina in DeepSpeed. The implementation has ∼7500 LoC.

6.1 Training

Lina's communication scheduler for training is deployed on all devices and runs a single thread. Since the communication scheduling is purely local in scope, no coordination is needed across the scheduler instances on different devices.

Communication scheduler. Each scheduler instance maintains a priority queue to schedule the micro-ops. The micro-op size is passed in as a hyperparameter. Lina uses the built-in APIs `chunk` and `cat` in LibTorch to partition the data in the token dimension. We avoid putting chunks from different gradients into the same micro-op to simplify the subsequent concatenation operation. Moreover, the scheduler stops launching allreduce micro-ops if the combining computation in backward pass, since this implies all-to-all is imminent. We pipeline all-to-all micro-op in the MoE layer. FFN is ready to start right after each all-to-all micro-op.

Expert packing coordinator. We embed a packing controller in the MoE model and it runs a single thread. Expert packing is dynamically adjusted after 10 training steps. In the forward pass, the controller records the completion times of all-to-all and FFN micro-ops. When FFN micro-ops are shorter than all-to-all, the controller starts to pack experts. First, we initialize the new process groups. Second, the controller inserts a one-time synchronous all-to-all to exchange expert parameters between packed devices that would be invoked at the upcoming iteration. Finally, multi-stream parallel execution is adopted for both forward and backward passes when more than one expert are hosted on a device.

6.2 Inference

Resource scheduler. The inference scheduler runs on a dedicated thread on device 0 of the cluster and manages resource scheduling. Each device saves the weights of all experts in

their host DRAM and the collected layer-wise expert popularity distribution using multiple `unordered_map`, one for each layer. If GPU memory is in shortage, a device only loads one expert and the profiled distribution of one layer at a time.

In phase one of scheduling, all relevant communication happens by piggybacking the information on the regular all-to-all to reduce overheads. For each MoE layer, each device appends the popularity estimation to the first all-to-all for device 0. The scheduler computes the new expert-device mapping and instructs each device which expert and how many to host via the second all-to-all. We also include necessary information to coordinate all-to-all of the next layer, including the list of devices with the same expert, and how many tokens each replica should handle to balance the load. Devices then swap in the expert weights for the next layer. All these procedures are pipelined with model computation.

In phase two, each device updates the actual expert popularity in a separate NCCL `send` to the scheduler. If no fine-tuning is required, the scheduler broadcasts a resume signal. This only creates a negligible overhead as the transfer size is tiny. Otherwise, Lina broadcasts the fine-tuned expert-device mapping. The model computation is blocked during phase two until the scheduler's command is received.

All-to-all coordination. In inference, Lina uses all-to-all with an unequal split. That is, the transfer size to each device in all-to-all does not need to be the same. Using unequal split all-to-all can save the overhead of initializing multiple process groups. A placeholder data pointer is passed to all-to-all if no tokens are directed to a certain device.

Expert packing. Expert computation is sequential on devices hosting multiple experts. Each device loads the experts one at a time to perform computation and move on to the next packed expert. In this manner, Lina avoids placing extra strain on the GPU memory. The second all-to-all is launched when the computation for all packed experts is completed. We set a maximum number of experts per device to control the overhead of swapping the weights.

7 Evaluation

We present the testbed evaluation results here.

7.1 Setup

Testbed setup. Our testbed has four worker nodes. Each node has 4 Ampere A100 GPUs with 40GB memory and is equipped with 100Gbps InfiniBand.

MoE models. We convert three common Transformer-based dense language models to MoE ones for training.

- Transformer-XL [20]: a 24-layer encoder model.
- BERT2GPT2 [49]: a 12-layer encoder-decoder model.
- GPT-2 [39]: a 12-layer decoder model.

Besides, we consider two inference tasks.

- Transformer-XL [20]: The inference task is text generation with Enwik8 [3] test set.
- BERT-Large [21]: a 12-layer decoder model. The inference task is translation using WMT En-De [8] test set.

All FFN layers in these models are converted to MoE layers. We vary the number of experts in an MoE layer from 2, 4, 8, to 16. We adopt top-2 gating in training and top-1 gating in inference following [23], i.e. $k = 2$ in training and $k = 1$ in inference

Metrics. We consider four metrics to evaluate Lina.

- Training step time: Time to complete one step of training.
- Inference time: Time to complete one batch of inference.
- All-to-all time: The completion time of all-to-all.
- MoE layer time: Time to complete one MoE layer of computation and communication.

In collecting these metrics we use PyTorch Profiler to obtain CUDA kernel execution time and GPU activities. Training results are averaged over 50 steps after a 10-step warm-up period. Inference results are averaged over the test set. Since the optimization introduced by Lina does not affect the precision of model parameters, model accuracy is unaffected and we omit its evaluation.

Training configurations. Lina's micro-op communication scheduler adopts a tensor partition size of 30MB, which can minimize the period blocked by all-to-all in most cases. Expert packing is launched at the 10-th step of each training task and is adjusted every four steps.

Inference configurations. Lina's resource scheduler runs on device 0. The path length l in popularity estimation is 3; the maximum number of experts packed on a device is 4.

Baselines. We use the vanilla DeepSpeed [2] as the Baseline. We also provide a comparison to the open-source version of Tutel [7], which performs similarly with DeepSpeed. We enable hierarchical all-to-all for both Lina and DeepSpeed and disable Random Token Dropping [50] introduced by DeepSpeed.

7.2 Training

We start with Lina's training performance. Note that Lina is evaluated when the expert packing decision is stabilized; all settings here use 2 experts per device as the best strategy except Transformer-XL with 16 experts, which uses 4 experts per device. The number of GPUs is equal to the number of experts per layer in both Baseline and Lina.

7.2.1 Overall Performance

Training step time. Figure 10 shows Lina's speedup in step time over Baseline and Tutel. All other aspects of the models stay the same (e.g. sequence length, hidden states dimension, etc.). Compared to Baseline (DeepSpeed), step time is reduced by an average of 1.37x and 1.47x for the 4- and 16-expert cases, respectively, and by an average of 1.71x and

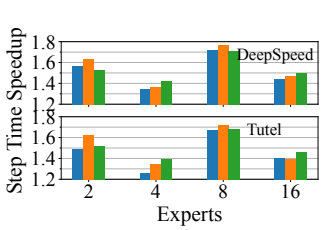


Figure 10: Speedup of training step time against two Baselines.

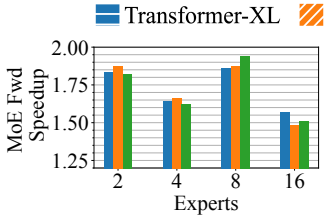


Figure 11: Speedup of MoE layer's forward pass completion time.

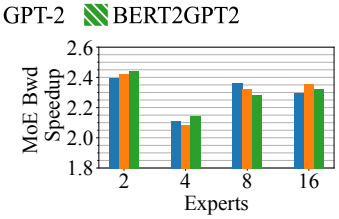


Figure 12: Speedup of MoE layer's backward pass completion time.

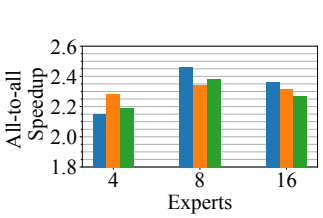
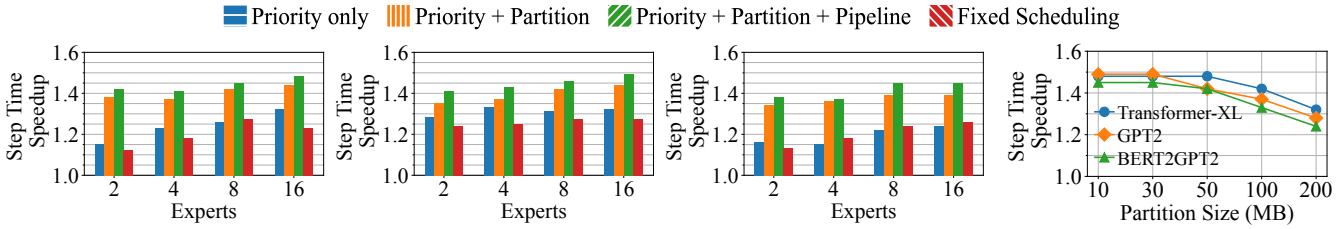


Figure 13: Speedup of all-to-all time in forward and backward pass.



(a) Transformer-XL. **(b)** GPT-2. **(c)** BERT2GPT2.

Figure 14: Training step time speedup over Baseline with different design choices of the communication scheduler.

Figure 15: Partition size increases from 10MB to 200MB in 16-expert models.

1.73x for 2- and 8-expert models, respectively. The 2- and 8-expert cases see more significant gains as Lina's packs two experts per device as mentioned before. The 2-expert case thus boils down to pure data parallelism without any all-to-all; the 8-expert models avoid inter-node all-to-all as our servers have 4 GPUs each. Lina's speedup over Tutel is slightly smaller than that of DeepSpeed. Thus in the following we only use DeepSpeed as the Baseline.

MoE layer time. We specifically seek to understand Lina's gain in MoE layers in both the forward and backward pass. As Figures 11 and 12 show, similar to step time, the gain in the 2- and 8-expert cases is the largest. The forward and backward pass of MoE layers in the 2-expert case are accelerated by 1.84x and 2.41x, and in the 8-expert case by 1.89x and 2.32x, respectively. Since backward pass in Baseline suffers from the interference of allreduce while the forward pass does not, the improvement in the backward pass is more significant. Average GPU utilization in the MoE layer for 16-expert cases is improved by at least 16% as the period blocked by all-to-all is minimized with Lina.

GPU utilization and memory usage. We measure the average GPU utilization GPU memory usage. We observe an average of 17.6% improvement in GPU utilization due to the efficient scheduling of Lina. Expert packing would lead to usage increase in GPU memory. The peak memory of BERT2GPT2 is increased by 19.5% while Transformer-XL and GPT-2 use up all the memory and apply DRAM-offloading to store the packed expert parameters.

All-to-all time. We then zoom in on all-to-all time in backward pass, where Lina prioritizes all-to-all and avoids concurrent execution with allreduce. Expert packing also reduces the all-to-all transfer size. Figure 13 shows an average speedup of 2.21x, 2.39x, and 2.31x in 4-, 8-, and 16-expert cases in all-to-all time, respectively.

Expert	Model	Pipelining Efficiency		
		w/o Packing	w/ Packing (Experts per Device)	
16	Transformer-XL	33%	86%	4
	GPT-2	36%	85%	2
	BERTGPT2	34%	79%	2

Table 3: Pipelining efficiency comparison with and without expert packing.

Expert	Model	Average GPU Utilization(%)		GPU Memory Peak Usage(%)		
		Baseline	Lina	Baseline	Lina	DRAM-offloading
16	Transformer-XL	66.2	83.4	72.1	100	✓
	GPT2	62.3	78.2	83.8	100	✓
	BERT2GPT2	63.5	82.5	74.3	94.2	✗

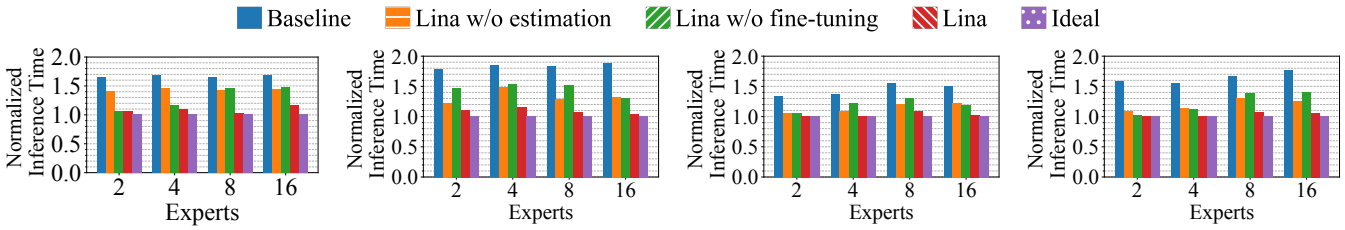
Table 4: GPU utilization and peak memory usage of 16-expert MoE models. GPU Memory Peak Usage is the ratio between the maximum usage and the total device memory. DRAM-offloading indicates if it is applied.

We also examine the pipelining efficiency between all-to-all and expert computation in Lina. We define the pipelining efficiency to be the fraction of non-idle time in the computation CUDA stream during the all-to-all duration. We calculate the pipelining efficiency of Lina before and after adopting expert packing in Table 3. The average improvement is 2.43x in 16-expert case, which also demonstrates the benefits of expert packing. The expert FFN micro-op time is thus closer to the all-to-all time. We find that two experts per device can achieve the best pipelining efficiency in most cases, justifying our settings mentioned before.

7.2.2 Communication Scheduler

We now present an in-depth analysis of Lina's priority-based micro-op scheduler, aiming to understand the benefit of each design choice. For fairness all experiments here are obtained without expert packing in Lina, i.e. one expert per device.

Tensor partitioning and pipelining. To justify our design, we incrementally add the key design choices to Baseline and see their corresponding gain: first priority scheduling, then tensor partitioning, and lastly pipelining. Besides, we



(a) Median inference time with Transformer-XL.

(b) 95%ile inference time with Transformer-XL.

(c) Median inference time with BERT-Large.

(d) 95%ile inference time with BERT-Large.

Figure 16: Median and tail inference time. We normalize the inference time with the ideal result. The median and tail inference time is the same in Ideal.

consider a fixed scheduling strategy where allreduce is always scheduled between pairs of all-to-all operations (i.e. two MoE layers) with tensor fusion enabled in PyTorch's `DistributedDataParallel` by default (same as Baseline).

Figure 14 shows the step time comparison. We make several interesting observations here. First, using priority brings about 10%–30% gain over Baseline in most cases, with an average of 24%. Priority scheduling in general presents more benefit when more devices and nodes are used in training. The main reason is that all-to-all's slowdown due to sharing bandwidth with allreduce is more severe as training scales out. Second, tensor partitioning significantly improves the benefit of prioritizing all-to-all: step time is reduced over Baseline by 1.36x, 1.36x, 1.41x and 1.42x in 2-, 4-, 8-, and 16-expert cases, respectively on average. On the other hand, pipelining's gain is limited as expected, since expert computation takes much less time than all-to-all without expert packing (recall §4.2). Overall, all three design choices can effectively reduce all-to-all's completion time.

We also observe that the relative benefit of priority scheduling and tensor partitioning is model-specific: GPT-2 enjoys much more gain from priority compared to tensor partitioning while the other two models do not exhibit such clear pattern. This is likely due to the degree of overlapping of all-to-all and allreduce: most allreduce can fit in between all-to-all operations in GPT-2, and as a result using priority scheduling alone is very beneficial.

Finally, the fixed scheduling strategy leads to the smallest gains in almost all cases. This is because (1) all-to-all still has to fair-share bandwidth with allreduce, and (2) tensors are not partitioned which aggravates the impact of allreduce. This demonstrates again the effectiveness of our design in prioritizing all-to-all with smaller tensors instead of using fixed heuristics that cannot opportunistically maximize efficiency.

Partition size. We also evaluate the impact of partition size on the communication scheduler. Figure 15 shows the step time of 16-expert models when we gradually increase the partition size from 10MB to 100MB. We find that a partition size beyond 50MB slows down Transformer-XL and BERT2GPT2 compared with 30 MB. As long as the period blocked by all-to-all is minimized, step time would be minimum. Therefore, for each model and setting, there are multiple optimal partition sizes. Ideally, the scheduler can more precisely control the operations with a smaller partition size. In practice, small partitions (below 10MB) may cause heavy transmission overhead in each micro-op and degrade the overall performance [37].

Overhead analysis. We provide a brief analysis of the overhead incurred by Lina's communication scheduler. First, the preprocessing and postprocessing, including tensor partitioning and concatenation, take an average 1.02% of the step time. Second, we measure the transmission overhead of micro-ops. We sum up running times of all the communication micro-ops and compare against those without partitioning in Baseline. The average completion time is lengthened by 1.7%.

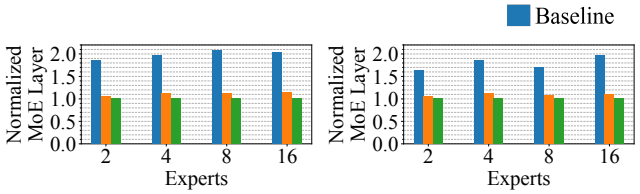
7.3 Inference

We then evaluate Lina's inference performance. Each experiment is repeated five times: two of which measure the end-to-end inference time, and the rest profile the different components with Profiler and collect statistics for overhead and estimation accuracy. This way the inference time is not affected by the profiling overhead.

7.3.1 Resource Scheduler

Inference time. Figure 16 shows the median and 95% inference time of Baseline and Lina. We also present the ideal inference time with a perfectly balanced load across devices in all MoE layers. This is obviously challenging to achieve with real-world requests. Thus we modify the gating network to constantly output a balanced expert selection to obtain this benchmark. We normalize all results to the Ideal value.

Lina's resource scheduler effectively balances the load among devices and achieve inference time close to Ideal. Compared to Baseline, median inference time is reduced by 1.54x and 1.45x for the 4- and 16-expert Transformer-XL, and by 1.36x and 1.46x for the 4- and 16-expert BERT-Large, respectively. The 95%ile inference time is reduced by 1.82x for 16-expert Transformer-XL and 1.68x for 16-expert BERT-Large. The reduction on tail inference time increases with more experts in a layer, because a wider MoE layer is more likely to present more skewed expert popularity, giving more room for Lina to optimize. Lastly, Lina's gap to Ideal can be explained for two reasons other than the overheads. First, Lina cannot perfectly balance load: the least popular experts are



(a) Transformer-XL. **(b)** BERT-Large.

Figure 17: 95%ile completion time of MoE layer.

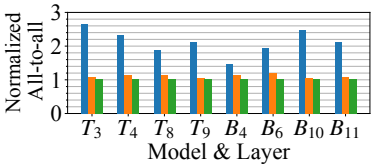


Figure 18: All-to-all time in 16-expert MoE. T is Transformer-XL and B is BERT-Large.

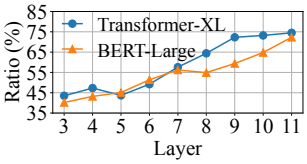


Figure 19: Estimation accuracy of 16-expert MoE.

randomly placed for example. Second, Lina starts to schedule from the fourth layer.

MoE layer and all-to-all time. With Lina, MoE layer time includes gate computation, phase two of scheduling, two all-to-all, and expert computation; phase one of the scheduling is largely overlapped with computation as explained in §6.2. The 95%ile MoE layer time is reduced by 1.87x and 1.77x in 8- and 16-expert Transformer-XL over Baseline and by 1.58x and 1.81x in 8- and 16-expert BERT-Large as in Figure 17. We also extract all-to-all time, which is a direct indicator of whether Lina balances load across devices effectively. We present the tail all-to-all time reduction of different layers in Figure 18. The average and maximum improvements are 1.96x and 2.50x over Baseline. These results confirm that Lina effectively balances the load of each device and all-to-all transfer size of each link.

Two-phase scheduling. We then evaluate the effectiveness of our resource scheduler's design. We consider separately Lina without estimation and without fine-tuning in order to understand their individual gains. Lina w/o estimation refers to scheduling using the actual routing decision computed by the gating network.

In Figure 16, we present the comparison of inference time for all schemes. Without estimation the median inference time is worsened by 24.0% and 18.6% for 16-expert Transformer-XL and BERT-Large in Lina. The scheduler works after the gating network and blocks all-to-all with the following computation until it completes. Thus the scheduling overhead manifests at each MoE layer, overweighing the additional gains brought by accurate popularity information. The tail inference time is less affected compared to the median, but still suffers without estimation.

Without fine-tuning, tail inference time is prolonged by 26.7% and 33.1% for 16-expert Transformer-XL and BERT-Large. This suggests that fine-tuning also plays an indispensable role when the estimation shows a large difference from the actual routing decision. For example, if the top-1 expert in the actual routing decision is estimated as an unpopular one packed with others, the Moe layer time would even be worse than Baseline. More discussions are presented in §7.3.2. The importance of fine-tuning depends heavily on estimation accuracy and number of expert in MoE layer.

Overhead analysis. We dissect the overhead of the resource scheduler, by considering the scheduling times of phase one and phase two separately. The scheduling time for both phases averages at ∼6.2ms since they share the same logic and coordination workflow. Yet, the overhead of phase one is largely overlapped with model computation. Though overhead of phase two with re-scheduling is more salient, it only kicks in for 23% of the cases on average, and is smaller than the idle time incurred by skewed expert popularity. The overhead of phase two without fine-tuning is merely 1.45ms.

Model	Path Length	Norm. Inference Time Median	Norm. Inference Time 95%ile	Fine-tuning (%)	Estimation Accuracy (%)
Transformer-XL	1	1.41	1.32	76.5	31.6
	3	1.16	1.04	25.7	60.4
	6	1.19	1.11	22.5	71.4
BERT-Large	1	1.34	1.35	71.3	28.3
	3	1.07	1.04	32.2	63.5
	6	1.09	1.11	27.1	66.0

Table 5: Lina's performance using different path lengths during estimation. Both models have 16 experts per layer. Inference time is normalized to Ideal.

7.3.2 Popularity Estimation

We now analyze Lina's popularity estimation method.

Estimation accuracy. We first examine the estimation accuracy across MoE layers. We resort to the same definition used in Lina's phase two scheduling: if the top-2 (recall $k = 1$ in inference) estimated experts are identical to the actual routing decision, we consider the estimation accurate. Figure 19 shows the accuracy for every MoE layer in two inference tasks. Overall, estimation accuracy is 58.41% and 54.16% for Transformer-XL and BERT-Large, respectively. The estimation accuracy is higher in the latter layers of the model, which is consistent with our observation in Figure 9. We also compare the complete popularity rankings given by the estimation to better understand its accuracy. Using 1000 random batches of the Transformer-XL model, we observe that errors usually happen at experts with a similar popularity. An average of 3.67 experts out of the estimation are incorrectly ordered. Therefore, the fine-tuning only requires little adjustment to the experts packed together. The effectiveness of Lina's estimation can be justified.

Sample path length. We also investigate the impact of sample path length l. The longer the sample path of expert selection is for making an estimation, the more accurate the result is. Table 5 shows Lina's performance degrades with $l = 1$, in terms of inference time, estimation accuracy, and the occurrence of phase two fine-tuning, compared with the default length of 3. Longer paths can elevate the estimation accuracy and further

Task	Dataset	Model	Norm. 95%ile Inference Time	Estimation Accuracy
Sentiment Analysis	IMDB Reviews [34]	BERT	1.08	64.4%
	Twitter [35]		1.11	62.3%
Translation (English)	WMT French [8]	T5 [40]	1.04	68.8%
	WMT Russian [8]		1.08	62.5%

Table 6: Lina's performance on different tasks and datasets. Inference time is normalized to Ideal. The path length is set to 3.

reduce the number of times of Lina's fine-tuning. However, due to the problem of a slower start, the reduction of inference time is not as noteworthy as the estimation accuracy. For a path length of 6, Lina shows a similar median and tail result as the performance with a path length of 3.

Generalizability. We proceed to evaluate how well Lina's popularity estimation approach can be generalized to different tasks. Table 6 shows the estimation accuracy of four tasks with different datasets. The 95%ile inference time can achieve at most 1.04x of the Ideal inference time and the estimation accuracy is at least 62.3%. Lina's estimation method relies on the patterns obtained from training stage. Therefore, it is tailored to each specific task and proves to be an effective approach to capturing the expert popularity prior.

8 Discussion

Parallelism in training. With the increasing scale of language models, the adoption of pipeline and tensor parallelisms has become essential [46]. Pipeline parallelism involves the use of blocking send and receive operations to transmit intermediate activations, while tensor parallelism utilizes blocking all-reduce operations to combine tensor partitions. Extensive research has been conducted on coordinating communication operations for dense models [7, 27, 52]. Lina focuses on sparsely-activated MoE with data and expert parallelism, which are orthogonal to existing work.

Estimation of expert popularity. The current estimation approach used by Lina relies on data collection during the training stage. While fine-tuning can assist in improving efficient expert placement decisions, an estimation method with improved accuracy and confidence would further reduce inference time. One potential approach is to leverage machine learning techniques to train a compact yet powerful model that can predict the expert selected by each token in every MoE layer ahead of time, when the requests are received.

9 Related Work

Existing MoE systems. Recent literature has proposed MoE-specific optimization techniques. DeepSpeed [41] enables distributed training for MoE models and leverages flexible combinations of parallelism strategies. It also introduces a novel MoE architecture called Pyramid-Residual MoE. PR-MoE applies experts only where they are most effective. Tutel [7] extends DeepSpeed and proposes an adaptive parallelism switching strategy specialized at MoE training tasks. It also includes a hierarchical all-to-all design to cope with the inter- and intra-node GPU topology for better efficiency. It is complementary with Lina.

FasterMoE [25] proposes a roofline performance model to analyze the end-to-end performance of MoE training systems. Guided by this model, they propose a dynamic shadowing approach that pulls popular expert parameters instead of sending tokens to the experts. They also design a topology-aware expert selection strategy that relieves network congestion by sending tokens to experts with lower latency.

Communication acceleration in distributed training. Our community has proposed several communication schedulers for generic distributed training [11, 17, 19, 24, 37, 48]. The objective is to better overlap the communication and computation operations in the backward pass and prioritize the communication of former layers over latter layers in the model. In Lina, we leverage the domain-specific insight that all-to-all should be prioritized over allreduce in MoE training, which is different from prior work. BytePS [28] proposes to reduce the communication traffic by utilizing the heterogeneous GPU/CPU resources in a training cluster. These acceleration techniques can be integrated into distributed MoE. Lina can also benefit from this idea, since more available bandwidth can be left to all-to-all operations.

10 Conclusion

We presented Lina, a new system that accelerates all-to-all in distributed MoE. Through a systematic analysis, we build Lina upon two key ideas: first to prioritize all-to-all over allreduce using tensor partitioning and pipelining to improve its bandwidth in training, and second to dynamically balance the workload with token-level expert selection pattern in inference. We implemented Lina over DeepSpeed and performed extensive testbed evaluation using A100 GPUs and 100Gbps InfiniBand to show that Lina significantly improves training efficiency and inference time.

Acknowledgment

We thank the anonymous ATC'23 reviewers and our shepherd Myoungsoo Jung for their constructive and valuable comments. We also thank the anonymous reviewers from NSDI'22 for their feedback that helped improve the paper. This work was supported in part by funding from the Research Grants Council of Hong Kong (N_CityU139/21, C2004-21GF, C7004-22G, R1012-21, R6021-20F, and 11209520), and from CUHK (4055138).

References

[1] ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/.

[2] DeepSpeed. https://github.com/microsoft/DeepSpeed.

[3] Enwik8. http://prize.hutter1.net/.

[4] NCCL. https://github.com/NVIDIA/nccl.

[5] PyTorch Distributed Data Parallel. https://pytorch.org/docs/stable/notes/ddp.html.

[6] PyTorch Profiler. https://pytorch.org/blog/pytorch-profiler-1.9-released/.

[7] Tutel. https://github.com/microsoft/tutel.

[8] WMT 19. https://github.com/facebookresearch/fairseq/blob/main/examples/wmt19/README.md.

[9] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, et al. Efficient large scale language modeling with mixtures of experts. *arXiv preprint arXiv:2112.10684*, 2021.

[10] Mandeep Baines, Shruti Bhosale, Vittorio Caggiano, Naman Goyal, Siddharth Goyal, Myle Ott, Benjamin Lefaudeux, Vitaliy Liptchinsky, Mike Rabbat, Sam Sheiffer, Anjali Sridhar, and Min Xu. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale.

[11] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM*, 2020.

[12] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent Shafey, Chandu Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml. In *Proc. MLSys*, 2022.

[13] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*, 2015.

[14] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[16] Tianlong Chen, Zhenyu Zhang, AJAY KUMAR JAISWAL, Shiwei Liu, and Zhangyang Wang. Sparse moe as the new dropout: Scaling dense and self-slimmable transformers. In *Proc. ICLR*, 2023.

[17] Yangrui Chen, Yanghua Peng, Yixin Bao, Chuan Wu, Yibo Zhu, and Chuanxiong Guo. Elastic parameter server load distribution in deep learning clusters. In *Proc. ACM SoCC*, 2020.

[18] Zewen Chi, Li Dong, Shaohan Huang, Damai Dai, Shuming Ma, Barun Patra, Saksham Singhal, Payal Bajaj, Xia Song, Xian-Ling Mao, et al. On the representation collapse of sparse mixture of experts. *Proc. NeurIPS*, 2022.

[19] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In *Proc. MLSys*, 2019.

[20] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[21] 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.

[22] 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 P Bosma, Zongwei Zhou, Tao Wang, Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. GLaM: Efficient scaling of language models with mixture-of-experts. In *Proceedings of Machine Learning Research*, 2022.

[23] 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.

[24] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proc. MLSys*, 2019.

[25] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. FasterMoE: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proc. ACM SIGPLAN PPoPP*, pages 120–134, 2022.

[26] Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Yong Li, Zhen Zheng, Xiaoyong Liu, Wei Lin, et al. Whale: Scaling deep learning model training to the trillions. *arXiv preprint arXiv:2011.09208*, 2020.

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proc. MLSys*, 2019.

[28] 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 *Proc. USENIX OSDI*, pages 463–479, 2020.

[29] 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.

[30] Aran Komatsuzaki, Joan Puigcerver, James Lee-Thorp, Carlos Riquelme Ruiz, Basil Mustafa, Joshua Ainslie, Yi Tay, Mostafa Dehghani, and Neil Houlsby. Sparse upcycling: Training mixture-of-experts from dense checkpoints. In *Proc. ICLR*, 2023.

[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] Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. Base layers: Simplifying training of large, sparse models. In *Proc. USENIX ICML*, 2021.

[33] Hanxue Liang, Zhiwen Fan, Rishov Sarkar, Ziyu Jiang, Tianlong Chen, Kai Zou, Yu Cheng, Cong Hao, and Zhangyang Wang. M³vit: Mixture-of-experts vision transformer for efficient multi-task learning with model-accelerator co-design. In *Proc. NeurIPS*, 2022.

[34] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proc. ACL*, 2011.

[35] Ibrahim Naji. TSATC: Twitter Sentiment Analysis Training Corpus. In *thinknook*, 2012.

[36] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The Case for Tiny Tasks in Compute Clusters. In *Proc. USENIX HotOS*, 2013.

[37] 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 *Proc. ACM SOSP*, 2019.

[38] Andrey Proskurin. DeepSpeed: Advancing MoE inference and training to power next-generation AI scale. https://www.microsoft.com/en-us/research/blog/deepspeed-advancing-moe-inference-and-training-to-power-next-generation-ai-scale.

[39] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[40] 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*, 2020.

[41] 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.

[42] 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. In *Proc. USENIX ATC*, 2021.

[43] 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. In *Proc. ACM NeurIPS*, 2018.

[44] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[45] Liang Shen, Zhihua Wu, WeiBao Gong, Hongxiang Hao, Yangfan Bai, HuaChao Wu, Xinxuan Wu, Haoyi Xiong, Dianhai Yu, and Yanjun Ma. Se-moe: A scalable and efficient mixture-of-experts distributed training and inference system. *arXiv preprint arXiv:2205.10034*, 2022.

[46] 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.

[47] 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*, 30, 2017.

[48] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In *Proc. MLSys*, 2020.

[49] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proc. EMNLP*, 2020.

[50] Zhewei Yao, Xiaoxia Wu, Conglong Li, Connor Holmes, Minjia Zhang, Cheng Li, and Yuxiong He. Random-ltd: Random and layerwise token dropping brings efficient training for large-scale transformers. *arXiv preprint arXiv:2211.11586*, 2022.

[51] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *Proc. USENIX ATC*, 2017.

[52] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *Proc. USENIX OSDI*, 2022.

[53] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. Mixture-of-experts with expert choice routing. *Proc. NeurIPS*, 2022.

[54] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. Designing Effective Sparse Expert Models. *arXiv preprint arXiv:2202.08906*, 2022.

[55] Simiao Zuo, Xiaodong Liu, Jian Jiao, Young Jin Kim, Hany Hassan, Ruofei Zhang, Tuo Zhao, and Jianfeng Gao. Taming sparsely activated transformer with stochastic experts. *arXiv preprint arXiv:2110.04260*, 2021.

SMARTMOE: Efficiently Training Sparsely-Activated Models through Combining Offline and Online Parallelization

Mingshu Zhai[◇] Jiaao He Zixuan Ma Zan Zong Runqing Zhang Jidong Zhai

Tsinghua University

Abstract

Deep neural networks are growing large for stronger model ability, consuming enormous computation resources to train them. Sparsely activated models have been increasingly proposed and deployed to reduce training costs while enlarging model size. Unfortunately, previous auto-parallelization approaches designed for dense neural networks can hardly be applied to these sparse models, as sparse models are data-sensitive and barely considered by prior works.

To address these challenges, we propose SMARTMOE to perform distributed training for sparsely activated models automatically. We find optimization opportunities in an enlarged space of hybrid parallelism, considering the workload of data-sensitive models. The space is decomposed into static pools offline, and choices to pick within a pool online. To construct an optimal pool ahead of training, we introduce a data-sensitive predicting method for performance modeling. Dynamic runtime selection of optimal parallel strategy is enabled by our efficient searching algorithm. We evaluate SMARTMOE on three platforms with up to 64 GPUs. It achieves up to 1.88× speedup in end-to-end training over the state-of-the-art MoE model training system FasterMoE.

1 Introduction

In recent years, a promising direction for deep neural network (DNN) design has been to increase model size. For example, pre-trained large models have shown extraordinary capabilities in natural language processing (NLP) tasks [1, 2, 13, 28].

As model size increases, training efficiency becomes increasingly important. From the system side, various parallel strategies (e.g., data [14, 18, 30, 33], pipeline [4, 10, 19, 24, 25], and tensor [35, 36, 38] parallelism) have been proposed to enable scalable distributed training. Furthermore, to hide underlying complex system details from researchers to allow them to focus on model design, automatic parallelization training systems [4, 24, 36, 41] have been proposed to automatically decide among various combinations of different parallel strategies to improve training efficacy. From the model design side, sparse architectures have been proposed to break the coherent relationship between model size and computation cost in DNN models with dense architectures. One of the most popular sparse models currently is Mixture-of-Experts (MoE) [12], which has significantly scaled up DNN models in many deep learning tasks, including natural language processing [3, 6, 15, 34], computer vision [11, 32], speech recognition [39, 40], and recommendation [22].

However, few efforts have been put into combining these two optimization directions. Existing training systems [8, 16, 38] typically adopt a specific expert parallelism to support distributed training of MoE models. Although expert parallelism mitigates the problem of high memory consumption of MoE models, training efficiency is affected. Several previous studies [9, 11, 17, 23, 27, 29] try to reduce the overhead of expert parallelism or combine expert parallelism with other parallel strategies, but all require special system expertise. Meanwhile, existing automatic parallelization training systems mainly target conventional DNN models with dense architectures. To improve both the user experience and the training efficiency of MoE models, it is indispensable to design an automatic parallelization training system for MoE models.

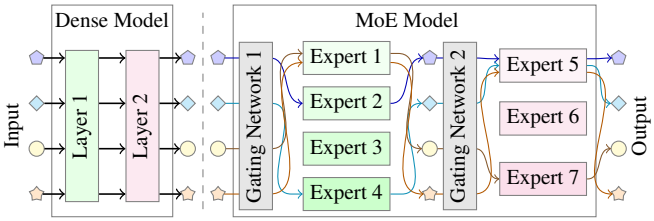


Figure 1: Dense Model Compared with MoE Model.

Figure 1 compares typical dense models with MoE models. In a dense model, the inputs are regarded as identical data to be processed by some layers. In an MoE model, the layers are replaced by multiple *expert sub-networks*. For each input, a special *gating network* is used to match it with the most

[◇]Tsinghua University, BNRist

suitable expert, and it is only processed by the selected expert. This leads to the dynamic and imbalanced property of MoE models, as the experts have different workloads. Some experts have to process more inputs than others, and this imbalanced situation is ever-changing across layers and iterations.

We identify the critical challenge of applying automatic parallelization techniques to MoE models due to the dynamic and imbalanced property, or being **data-sensitive**. While the training cost is fixed in dense models over any input, MoE models behave differently over different data, layers, and training steps. Because the gating network dynamically matches training inputs with experts, the workload of experts may vary a lot, resulting in varying costs for computation and communication. Unfortunately, current automatic parallelization approaches fail to efficiently deal with data-sensitive training of MoE models due to the following two limitations.

Limited Optimization Space. Being data-sensitive makes previous approaches of parallelism combinations perform differently and introduces more space and opportunities for optimizations. Compared to dense models, heterogeneous workloads on different expert sub-networks in MoE training lead to a much larger combination space of parallelism. We find that with the workload variance in mind, there are more opportunities of optimizing training performance. However, existing works [36, 41] assume that the workloads on sub-networks are homogeneous, and exclude many potentially faster candidates from their space for hybrid parallelism.

Large Searching Overhead. For data-sensitive models, the optimal execution plan changes frequently. However, for previous data-insensitive systems, the workload is static and can be determined by the model structure before training. Therefore, they adopt time-consuming algorithms, e.g., dynamic programming [24, 36] or integer linear programming (ILP) [41], to search for optimal execution plans. These algorithms commonly take minutes or even hours, only feasible to be performed offline. However, optimal execution plans for the dynamic workload should be identified between iterations that commonly take less than one second.

To address these challenges, we propose SMARTMOE, an automatic parallelization training system for sparsely activated models. We explore the space of hybrid parallelism with awareness of heterogeneous workloads, where more potentially faster candidate parallel strategies are included. To sustain high efficiency during the dynamic and imbalanced MoE training process, we propose a two-stage solution for parallelization. Based on a **static pool** that consists of mutual-convertible parallel strategies constructed offline, fast **dynamic adaption** is performed within the constructed pool at runtime to select the strategy that fits the current workload.

In the offline stage, we create a pool of strategies that guarantees good inherent performance and low switching overhead at runtime. Also, we design a workload-aware performance model to estimate the performance of the data-sensitive models without actually training them so that an optimal pool can be constructed ahead of training.

In the online stage, we develop light-weight algorithms to find the optimal parallel strategy for the current workload within the selected pool. The algorithms are performed periodically at runtime to determine whether we should employ a new parallel strategy, considering factors including switching cost and searching overhead.

We evaluate SMARTMOE on three different clusters with up to 64 GPUs. Results show that SMARTMOE achieves up to 1.88× speedup in end-to-end training compared with the state-of-the-art MoE model training system FasterMoE [9].

In summary, we make the following contributions:

- We enlarge the combination space of hybrid parallelism for data-sensitive models, enabling more potential to optimize training performance.
- We propose a two-stage adaptive auto-parallelization approach that performs hierarchical optimizations both offline and online.
- We introduce the awareness of workload to performance modeling, enabling performance prediction of training the data-sensitive models.
- We develop fast algorithms that can find optimal parallel strategies within a pool at runtime.
- We implement these techniques into an end-to-end MoE training system, SMARTMOE, and achieve up to 1.88× speedup over FasterMoE [9].

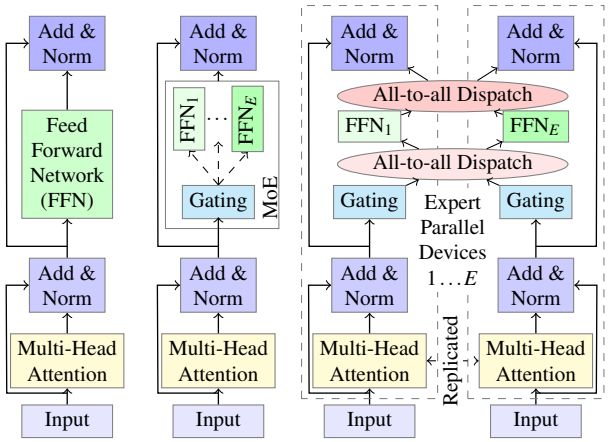
The rest of this paper is organized as follows. §2 introduces background and our motivation. §3 presents an overview of SMARTMOE. §4 introduces an enlarged space of hybrid parallelism for MoE model training. §5 discusses the scope of pool among the space of data-sensitive hybrid parallelism, and demonstrates our estimation-based approach of performance modeling. §6 illustrates our adaptive automatic parallelization methods. §7 evaluates SMARTMOE. More related works are described in §8, and §9 concludes this paper.

2 Background and Motivation

2.1 MoE Model and Expert Parallelism

The Mixture-of-Experts (MoE) was proposed decades ago [12] and applied to DNN models in recent years. It has been proven to be helpful in improving model accuracy in many deep learning tasks, including nature language processing [15], computer vision [11], speech recognition [39, 40] and recommendation [22]. In this paper, we focus on sparsely-gated MoE [34] models, the most widely used MoE technique, with instant demand for efficient distributed training.

The MoE technique is currently the most feasible way to enable the parameter size of a model and its computational cost to be scaled independently. A model can increase the number



(a) A Transformer Block (b) An MoE Transformer Block (c) Expert Parallelism for the MoE Transformer Block

Figure 2: An Example of MoE Model and Expert Parallelism.

of parameters by applying MoE, while keeping the floating-point operations (FLOPs) of one training iteration almost identical. For example, Figure 2 shows the model structure of the transformer model extended by MoE. A feed-forward network (FFN) is regarded as an expert, and the model contains multiple experts which are sparsely activated. A trainable gating network is added to dynamically match training samples with suitable experts. As each training sample is sent to a certain expert, which equals the original FFN in size, FLOPs required to train over the sample remains similar. Meanwhile, numerous experts can be employed in one MoE layer, greatly increasing the number of parameters.

Distributed training becomes a must to train MoE models, as the model is so large that it cannot be held in the memory of any single device. To support the distributed training, GShard [15] designs a specific method of parallelism for MoE models, namely **Expert Parallelism (EP)**. In fact, it is a combination of Data Parallelism and Tensor Model Parallelism specialized for the MoE scenario. As shown in Figure 2(c), the model is split up across the dimension of the experts' indices, and the input and output features are split along sample dimension. All-to-all communication is performed to dispatch the input samples to the desired expert models and put the output back to its original location, e.g. re-arranging words into sentences in language models.

Dynamic routing is the most unique feature of the MoE training workload. A trainable gating module dynamically assigns tokens to different experts in every iteration for every MoE layer, according to the input data. Therefore, the training workload varies at different layers and iterations. This dynamic nature of the MoE models makes it much different from a traditional neural network in distributed training.

2.2 Hybrid and Automatic Parallelization

Listed below are three common ways of parallelism to train typical dense deep neural networks.

Data Parallelism (DP). Each worker stores a complete copy of parameters, and the training samples assigned to each worker are different. Forward and backward computation are completed independently on each worker. Gradients on different workers are aggregated before being used in the optimization of the model. DP incurs significant memory and communication overhead as the model gets larger, because all the parameters are replicated and synchronized in every iteration. Some approaches [30, 37] reduce the memory footprint by splitting up the replicas, but the communication overhead is inevitable.

Pipeline Model Parallelism (PP). The model is divided into multiple stages with sequential data dependency. Each worker stores the parameters of its corresponding stage. The first worker reads batches of the training data, and workers with adjacent stages exchange intermediate results for forward or gradients for backward computation. To be efficient, PP has to have evenly distributed stages and bubble-free schedule, intensively studied by prior works [4, 10, 24, 25].

Tensor Model Parallelism (TP). Single operators of a model are partitioned across multiple workers. Each worker stores a part of the parameters of the operators and conducts part of its computation, e.g. a tile of a matrix. TP of different operators needs to be designed specifically by experts, and the partitioning method is critical to distributed training performance. Megatron [35] provides the best practice of TP on transformer models. Other works [36, 38] explores unified representation of TP and automatic generation of the most efficient partition.

To improve distributed training performance, **Hybrid Parallelism** is introduced, which combines a few of the above parallel strategies to better fit specific models and particular training hardware. We call an instance of hybrid parallelism a **parallel execution plan**. Given a model and a hardware specification, there can be multiple parallel execution plans, since multiple parallel strategies are available. For example, Megatron-v2 [26] achieves high-performance distributed training by expert-designed hybrid parallel execution plans, but only for transformer-based models.

Moreover, **automatic parallelization** is desired to make high-performance hybrid parallelism available to model developers with less expertise in distributed systems, Alpa [41] categorizes parallelism into inter-layer (PP) and intra-layer (DP and MP) levels, and automatically generates hybrid parallel execution plans by hierarchically optimizing over both levels. However, it is very time-consuming for current approaches to generate an optimal parallel execution plan, due to the lack of performance models and their excessive searching algorithms. Minutes, or even hours, are taken to generate an execution plan that may only cost milliseconds or seconds for an iteration.

In the end, we summarize three key challenges for any automatic parallelization training system.

Space of Hybrid Parallelism. Hybrid parallelism means combining multiple different parallel strategies into one ex-

ecution plan. Specifically, the hybrid of any two different strategies may involve complicated adaption, and introduce variance in performance. The more parallel strategies a system can handle, the more opportunities exist to find a faster execution plan.

Performance Modeling. Performance modeling helps explore a huge hybrid parallelism space efficiently, as it is infeasible to measure the cost of every possible execution plan without actually running it.

Besides, beyond being accurate as a basic requirement, a good performance model shall be giving extra information, or guidance, that can provide better understanding of the performance, and indicate the direction of generating a better execution plan.

Searching Algorithm. The huge space of hybrid parallelism shall be explored adequately to find an optimal or near-optimal execution plan. However, for large-scale model training, it is even unacceptable to enumerate every possible candidate. The algorithm's efficiency in finding a near-optimal execution plan is appreciated, primarily when performed frequently over different configurations.

2.3 Challenges of Automatic Parallelization for MoE Models

The aforementioned issues are even more challenging with MoE models. Unlike typical dense models, the dynamic nature of the MoE training workload makes them more complicated and invalidates existing automatic parallelization approaches. We use an example of MoE training in Figure 3 to explain the challenges.

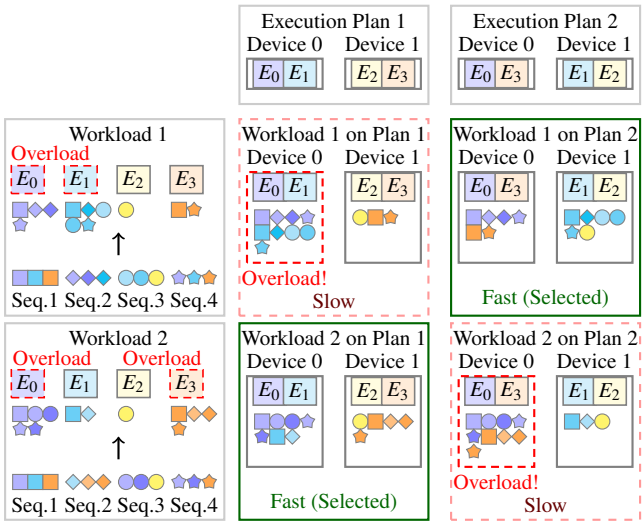


Figure 3: Example MoE Training Workloads and Related Parallel Execution Plans.

Larger Space of Hybrid Parallelism. The dynamic nature of MoE leads to heterogeneous workloads on different expert sub-networks, resulting in a larger space of hybrid parallelism for MoE models than that for dense models. For a 4-experts MoE layer, two possible execution plans are shown at the top of Figure 3. In the conventional view, these two plans are identical, as both devices contain two experts, so their memory consumption and computation costs are identical. However, the dynamic workload of MoE training can lead to computational bottlenecks occurring on different experts during the training process, resulting in performance differences between the two example execution plans. For example, execution plan 1 suffers from a load-imbalance problem over training workload 1, while execution plan 2 does not.

State-of-the-art MoE training systems [9, 11, 17, 23, 27, 29] fail to support execution plan 2, because none of them study the order to place expert sub-networks on multiple devices. State-of-the-art auto-parallelization systems [36, 41] also ignore execution plan 2, as both execution plans are treated as the same in their algorithms. To avoid missing efficient execution plans, exploring the space of hybrid parallelism with an awareness of the imbalanced dynamic workload of MoE models is necessary.

Workload-Aware Performance Modeling. Conventional performance modeling approaches only use model structure and hardware information to estimate the performance and lack consideration of the workload. However, the dynamic nature of MoE causes a strong relationship between ever-changing training workloads and efficiency, invalidating conventional performance modeling approaches of DNN operators. Looking back to the example in Figure 3, the efficiency of execution plans is different under two training workloads. Execution plan 1 becomes better in workload 2, though it is slower in workload 1. There is a strong connection between the efficiency of an execution plan and the training workload, being a unique feature of MoE models.

Current auto-parallelization algorithms [36, 41] search for an efficient execution plan ahead of training, lacking consideration of the dynamic workload. A dynamic workload-aware performance modeling approach is demanded to provide accurate estimation for the MoE training scenario.

Adaptive Automatic Parallelization. For data-insensitive dense models, parallelization is performed once before training to generate an optimal execution plan. However, since no single execution plan can fit all workloads in MoE training, using a fixed execution plan cannot be efficient throughout the MoE training process. Adaptive automatic parallelization is demanded, which employs runtime execution plans searching and switching to maintain high efficiency through the training process. Ideally, the searching and switching procedure is so efficient that a training system can change the execution plan at every iteration to achieve ultimately high performance.

However, it is hard to achieve the ideal case due to the following issues. First, state-of-the-art systems use time-consuming algorithms such as integer linear programming for an optimal execution plan, which cannot fit the time limit. Second, switching between different execution plans can be

expensive because of the high overhead of parameters exchange through inter-device links.

In existing distributed training systems [4, 23, 24, 26, 41], they select an execution plan before training without the ability of runtime execution plan adaption. Besides, the searching overhead of current auto-parallelization algorithms [36, 41] is too high to be used at runtime. Light-weight methods to generate and switch between execution plans according to the dynamic workload are appreciated for an MoE training system.

3 Overview

We propose SMARTMOE, an automatic parallel training system for sparsely activated models. Previous automatic parallelization systems only search for the optimal execution plans before training, while SMARTMOE uses a two-stage approach. Beyond prior works that generate optimal execution plans based on *model* architecture and *hardware* specification, we take the **workload** into account for data-sensitive models. We introduce an enlarged space for hybrid parallelism in §4. It introduces more opportunities for better training efficiency. With this enlarged space, SMARTMOE supports efficient execution plans for data-sensitive models. Moreover, to achieve efficient workload-aware parallelization, we split the process of automatic parallelization into two stages, performed offline and online, respectively. Figure 4 presents an overview of our two-stage algorithm.

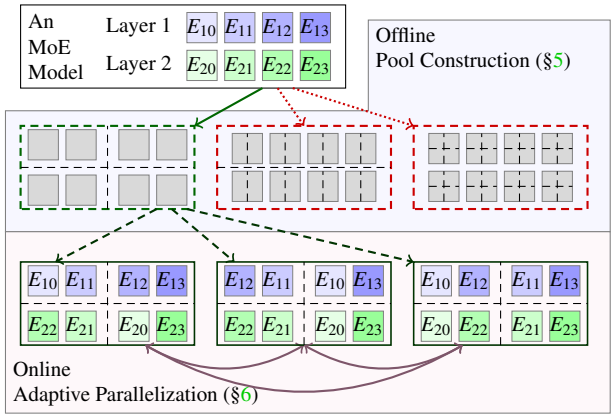


Figure 4: Overview of Two-Stage Auto-Parallelization.

Offline Pool Construction (§5). The imbalanced workload introduces potentially faster candidate execution plans. Among them, some pairs of execution plans are identified to be much more expensive to switch between than others and infeasible to be performed online. Therefore, SMARTMOE clusters a group of execution plans as a **pool**, among which the switching cost is moderate. SMARTMOE constructs a promising pool ahead of training, while keeping the ability of online adaption within the constructed pool.

An optimal pool has to be aware of the workload. We design a data-sensitive performance model to help construct good pools among numerous candidates, utilizing model specifications to estimate the workload without using statistics of actual workload. As the time limit is relatively relaxed before training, we exploit searching algorithms of conventional methods with our performance model to construct the pool.

Online Adaptive Parallelization (§6). A pool commonly contains an exponential number of execution plans to be selected, but online decisions should be made in milliseconds. We develop light-weight algorithms to meet the time limit. The algorithms can be intensively performed to ensure the execution plan fits the current workload and quickly find faster ones from the pool if available.

Online adaptive execution plan switching can only be practical considering the overhead itself. We find it a trade-off between the high efficiency of an execution plan and the latency to switch to it. We utilize temporal locality in the ever-changing workload of MoE model training to adjust parallel strategy at the proper time and achieve overall performance improvement.

4 Enlarged Space for Hybrid Parallelism

In most of the previous MoE model training systems, only expert parallelism is used for MoE models. A representative system of this class is FasterMoE [9], which focuses on optimizing expert parallelism. Fewer previous works support hybrid parallelism for MoE models. A notable system of this class is BaGuaLu [23], which combines expert and data parallelism to train MoE models on a full-scale supercomputer. Different from previous works, SMARTMOE supports hybrid parallelism for MoE models comprehensively.

SMARTMOE supports an arbitrary combination of data and tensor, pipeline and expert parallelism, beside a barely studied aspect of parallel strategies, namely **expert placement**. With the help of this enlarged space for hybrid parallelism, SMARTMOE could cover more efficient parallel execution plans than previous works.

Table 1: Configuration of Expert Slot.

Parallel Strategy	Capacity	# Slots	# Layers
Expert	1	E/N	L
Expert+Data	1	$D \times E/N$	L
Expert+Tensor	$1/T$	$T \times E/N$	L
Expert+Pipeline	1	$E/(N/P)$	L/P

SMARTMOE supports an arbitrary combination of existing parallelism using the concept of **expert slot**. An expert slot is a basic unit to store parameters of an expert sub-network on workers. To specify a parallel strategy for an MoE layer, the configuration of expert slot for every worker should be determined. Formally, we use three attributes to represent a configuration of expert slot:

- The capacity of each slot. It should be a fraction between 0 and 1, as each slot can store the partial or whole of an expert sub-network.
- The number of slots for each worker. It should be positive, as each worker contains one or more slots.
- The number of MoE layers for each worker. It should be a positive integer, depending on the model structure.

These attributes are powerful to represent classic parallel strategies and their combinations. Suppose a model contains L MoE layers and E experts in each layer. The training is done on a N workers cluster. D, T, P represent ways of data, tensor, and pipeline parallelism respectively. We showcase how to set attributes for different parallel strategies in Table 1. We also provide a concrete example in Figure 5, where $(L, E, N) = (2, 4, 4)$. In Figure 5(c), the setting is $(D, T, P) = (2, 1, 1)/(1, 2, 1)/(1, 1, 2)$, respectively. It is worth noting that only combinations of at most two parallel strategies are shown in the above examples. SMARTMOE can instantiate their arbitrary combinations, as they can be represented as specific configurations of the expert slot.

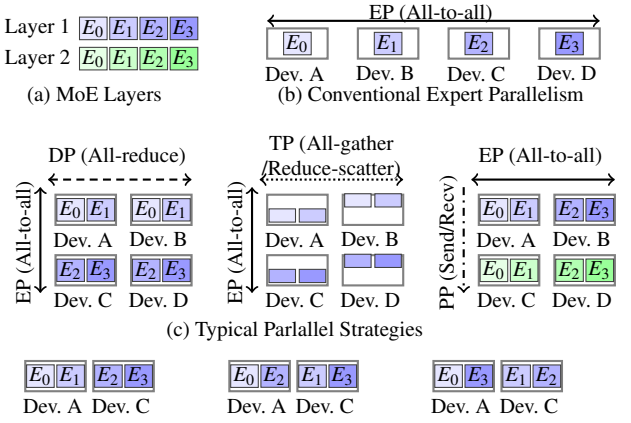


Figure 5: Hybrid Parallelism for MoE Models.

SMARTMOE explores a new parallel strategy, **expert placement**. The expert placement plan is an essential but understudied aspect of MoE model training. An expert placement plan refers to a mapping from expert sub-networks to expert slots, as shown in Figure 5(d). When the number of slots on each workers is more than one, an expert placement plan could influence performance because of the imbalanced workload. Recall the example in Figure 3. Two execution plans only differ in expert placement, but their efficiency significantly varies due to the imbalanced workloads. To the best of our knowledge, previous MoE training systems do not study the difference among different expert placements, i.e., expert sub-networks are stored in multiple slots in serial order.

As Figure 5 suggests, besides supporting combinations of existing parallelism (Figure 5(b,c)), SMARTMOE can explore various expert placement plans (Figure 5(d)) for better performance. In contrast, existing MoE training systems only work on limited cases of our space. For example, FasterMoE [9] focuses on runtime optimization of expert parallelism (Figure 5(b)). BaGuaLu [23] and DeepSpeed-MoE [29] adopt specific hybrid parallel strategies in Figure 5(c).

5 Offline Pool Construction

5.1 Design Principle of a Pool

A pool is a sub-space of hybrid parallelism that contains multiple execution plans with some parallel strategies fixed and leaving the others variable. The pool remains unchanged throughout the process of distributed training. For example, a pool can be a condition that over multiple nodes with multiple GPUs, pipeline parallelism shall be used across nodes, which is fixed. And any parallelism, such as data or model parallelism, can be used among GPUs within a node. SMARTMOE constructs a good pool before training and switches execution plans at runtime within this constructed pool.

The critical challenge of defining the scope of pools is dividing parallel strategies into two categories: fixed and variable. The fixed parallel strategies have inherent latency of execution plans, which are almost unaffected by dynamic workload. In contrast, runtime switching on the variable parallel strategies is necessary to fit the current workload. In addition, the overhead of runtime switching should be balanced with the performance gain.

In SMARTMOE, we define a pool as **a group of execution plans where expert placement is the only variable parallel strategy.** Expert placement, i.e., mapping from experts to devices, is found to be non-trivial due to the heterogeneous workload on experts. We identify that hybrid parallel strategies for typical dense models steadily impact performance. In contrast, expert placement could influence performance significantly when the workload changes dynamically. So, in the offline pool construction stage, SMARTMOE searches for an excellent combination of typical parallel strategies, while the expert placement plan is variable to be switched online.

This definition of the pool has two main advantages. First, the space of candidate execution plans has enough flexibility for online adaption. The number of possible expert placement plans is increased exponentially when there are more expert slots for each worker. It is promising to find a suitable execution plan according to the current imbalanced workload (Detailed in §6.1). Second, these candidate execution plans are mutual-convertible with minor overhead, avoiding introducing much runtime overhead. As they have the same configuration of expert slots, there is no need for memory allocation or release when switching. The switching overhead is only caused by parameter exchange between workers, and it is possible to maintain a moderate communication overhead (Detailed in §6.2).

5.2 Workload-Aware Performance Modeling

In the offline stage, SMARTMOE constructs a good pool before training. A critical step is using a performance model to estimate the efficiency of different pools. However, designing an accurate performance model offline is challenging because the performance of an execution plan is strongly related to the dynamic training workload, which cannot be obtained before training.

We introduce a method to estimate the training workload to address this challenge. Specifically, it estimates the distribution of expert selection, i.e., the outputs of gating networks, before training. So, it realizes workload-aware performance modeling of MoE layers without actual statistics of the training workload. With its help, we can accurately estimate the computation and communication costs of candidate pools before training. Finally, we apply the data-sensitive performance model over the candidate pools and enumerate the search space before initiating distributed training.

Semantics of the gating network guides the workload estimation. Based on the specific algorithm of any given gating network, the maximum amount of workload for any expert can be calculated without actually training the model. This upper bound of workload is commonly close to the actual workload and is the bottleneck of the training process. Therefore, using the maximum possible amount of workload, we can accurately predict the performance of a pool.

We take the most common *Choose Top-K along Experts Axis* gating approach as our instance, and our modeling method is applicable to many others [5]. The key idea is to understand the design of gating networks for accurate estimation of computation and communication costs. We classify state-of-the-art MoE gating networks into two classes and explain our estimation approach for them respectively.

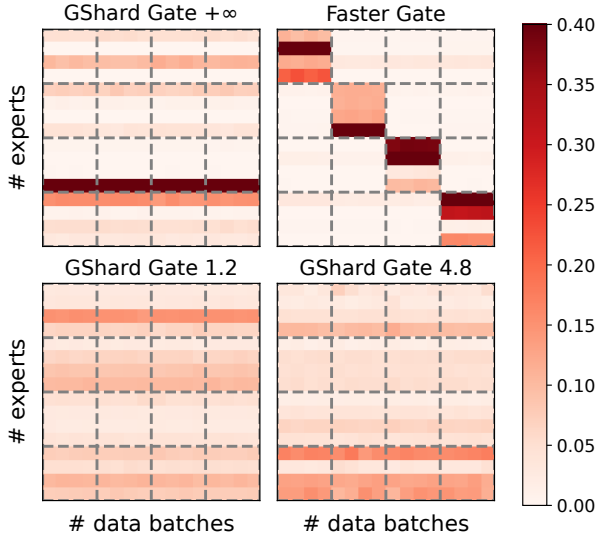


Figure 6: Expert Selections under Different Gating Networks.

One is load-balanced gating networks, which ensures a balanced workload on experts. GShard [15] gate, a commonly used gating method, represents this class. The critical design of the GShard gate is the *capacity factor*, which limits the proportion of training samples assigned to remote experts (i.e., experts placed in different devices with input samples). We visualize expert selections of GShard gate from a real training process in Figure 6. The capacity factor is set to 1.2, 4.8, and $+\infty$ (i.e., no capacity limit). Controlled by different capacity factors, the proportion of training samples assigned to the overloaded experts varies. Despite the variation, there is a definite load upper bound of the most overloaded expert, being the bottleneck of the whole layer. Therefore, we use that upper bound to estimate the performance of execution plans.

Another is topology-aware gating networks, which limit the size of cross-nodes communication in the all-to-all dispatching stage. Two state-of-the-art MoE models [7, 9] use topology-aware gates. In Figure 6, *Faster Gate* proposed by FasterMoE [9] is visualized. The example cluster has 16 devices on four nodes with a fat-tree network topology. To avoid suffering low bandwidth across nodes, it prefers to assign training samples to experts within the same node, as can be seen in the figure that most expert selection is in 4×4 blocks on the diagonal. We can estimate expert selection considering its hierarchical gating algorithm. As it also uses hardware specifications available to us, we follow its algorithm to compute the maximum possible communication volume and computation workload for each device and adopt these data for our performance prediction.

Generally, although the actual expert selection is unreachable, hyper-parameters of these gating networks can be used to depict the distribution of expert selection. The computation overhead of the most overloaded expert can be estimated according to the capacity factor, and the communication overhead of device pairs can be estimated according to constraints provided by topology-aware gating networks. After obtaining the distribution of expert selection, we adopt an existing performance model [9] to estimate the performance of execution plans. It is worth noting that the original model has to take the current workload information captured at runtime as its input. However, in SMARTMOE, we apply it to the estimated workload. With the help of our estimating method and the performance model, we apply exhaustively offline searching for a pool with the best-estimated performance.

6 Online Adaptive Parallelization

6.1 Light-Weight Searching

Adaptive parallelization is required for the MoE model to keep it on an optimal execution plan considering its current workload. The challenge of runtime adjustment is a much stricter time limit of searching overhead, as a single iteration

of an MoE layer typically takes only tens of milliseconds. Existing works fail to meet the time requirement of MoE models because they commonly apply time-consuming algorithms such as integer linear programming (ILP). In practice, the overhead of these methods is orders of magnitude greater than the latency of a single iteration. They are infeasible for MoE models because they shall be intensively performed at runtime to ensure the execution plan fits the current workload.

Defining a pool as only expert placement plans can be adjusted at runtime is a good equilibrium point: workload-guided expert placement plans effectively improve training efficiency, while switching between expert placement plans does not introduce much communication overhead. Based on the pool constructed offline, we make minor adjustments to fit the current workload. To keep moderate overhead, we design light-weight searching algorithms to transfer within the pool among execution plans. We invent a greedy algorithm to replace time-consuming methods such as ILP, which is intensively performed to ensure the execution plan fits the current workload.

Motivated by our observation in Figure 3, it is critical to generate a dedicated expert placement plan which fits the current workload. We formalize the expert placement problem as follows. Supposed that there are E experts and N devices, C_i training samples are assigned to expert i by the gating network. We need to decide the placement of each expert, where the placement of expert i is denoted as $P_i (i \subseteq \{0,1,\ldots,N-1\})$. The optimization goal is to minimize eq. (1), where $\frac{C_i}{\|P_i\|}$ denotes the number of training samples processed by each replicate of expert i, and overall training latency is determined by the most overloaded device j.

$$\max_{0 \le j < N} \{ \sum_{0 \le i < E, j \in P_i} \frac{C_i}{\|P_i\|} \} \tag{1}$$

Algorithm 1 Greedy Expert Placement

```
1:  function ExpertPlacementGreedy(E, N, C[E])
2:      samples[0...N] ← 0        ▷ current samples per device
3:      experts[0...N] ← 0        ▷ current experts per device
4:      P[0...E] ← ∅              ▷ placement of experts
5:      for i, C_i ∈ DescendingSort(C) do
6:          T_min ← ∞
7:          for all j do          ▷ decide the placement
8:              if experts[j] < E/N and samples[j] < T_min then
9:                  T_min ← samples[j]
10:                 p ← j
11:         P[i] ← P[i] ∪ p
12:         samples[p] ← tokens[p] + C_i
13:         experts[p] ← experts[p] + 1
14:     return P[0...E]
```

We propose a light-weight greedy approach in Algorithm 1. An intuitive idea is to avoid placing overloaded experts on the same device. Therefore, Algorithm 1 decides each expert's placement in the order of amounts of computation on experts. To avoid increasing memory overhead on certain devices, the number of experts placed in one device is limited to $\frac{E}{N}$. This algorithm's computational complexity is $O(NE)$, which is light enough for runtime searching.

We also propose a more accurate but costly dynamic programming approach. The state of dynamic programming is defined as $F(i,S)$, in which i denotes the number of devices that are fully used, S is a subset of N experts to denote which experts have been placed. And F represents the minimal size of the most overloaded device in the first i devices. Equation (2) shows the transfer function between states, which enumerates experts placed on device i for a minimum cost. In this dynamic programming approach, the number of states is $O(N \times 2^E)$, and $O(2^E)$ states are enumerated in one transfer, resulting in total computation complexity $O(N \times 4^E)$. This approach is guaranteed to find an optimal solution.

$$F(i,S) = \min_{S_0} \{ \max \{ F(i-1, S_0), \sum_{e \in S - S_0} C_e \} \} \tag{2}$$

To leverage the advantages of the two algorithms above, we design Algorithm 2 to combine them. The problem of placing E experts into N devices is divided into two steps: in the first step, E experts are placed into M virtual devices using the greedy algorithm; in the second step, M virtual devices are placed into N devices using the dynamic programming algorithm. The computation complexity of this hybrid algorithm is $O(ME + N \times 4^M)$. The number of virtual devices M is a tunable parameter, which controls the overhead of the searching algorithm. For example, modern clusters usually have tree topology: there are multiple devices within a compute node, and multiple nodes are working together. In this setting, the M can be set as the number of devices within a node, in which the greedy algorithm is used across nodes, and the dynamic programming algorithm is used within a node.

Algorithm 2 Hybrid Approach of Expert Placement

```
1:  function ExpertPlacementHybrid(E, N, C[E])
2:      M ← DEVICES_PER_NODE      ▷ tunable parameter
3:      P_0[E] ← ExpertPlacementGreedy(E, M, C[E])
4:      P[0...E] ← ∅
5:      for i ← 0, ..., M do      ▷ Use dynamic programming
        within a virtual device
6:          S ← {e | i ∈ P_0[e]}
7:          P_S ← P(F(M, S))      ▷ Get placement of DP state S
8:          P ← P ∪ P_S
9:      return P[0...E]
```

By properly tuning M according to hardware configuration and searching time limit, these algorithms can work together with minimum overhead and maximum effect.

6.2 Efficient Adaptive Training

The goal of the online stage is to obtain performance gain by applying an execution plan that fits the current workload while keeping moderate switching overhead between execution plans. A few configurations of SMARTMOE are important to ensure performance gain. We currently tune them manually by heuristics, as discussed below.

Threshold of Switching Overhead. Switching from the current expert placement plan to a newly generated one introduces non-negligible all-to-all communication overhead. However, in practice, some newly generated expert placement plans only make slight advancements over the current one while introducing much communication overhead. To address this problem, we set a tunable threshold to filter out new expert placement plans with minor improvement in eq. (1). In addition, if two new plans have the same latency in eq. (1), SMARTMOE chooses the one similar to the current plan to minimize the switching overhead.

Frequency of Online Searching. Although online searching fits the expert placement plan with the current workload, searching and switching costs extra time. One extreme case is to perform searching at every iteration, which can always fit the current workload with the best execution plan but introduces too much overhead. However, if the search interval is too large, the training throughput usually decreases as the workload changes. Fortunately, as a neural network, parameters of the gating network change slightly in adjacent iterations, which leads to a gradual change in the distribution of expert selections. Thanks to this temporal locality, we can conduct runtime searching every several iterations. Figure 12 reveals the trade-off between searching frequency and training performance in real model training. We currently select an appropriate frequency by experimentation.

Frequency of History Collecting. Our online searching algorithm depends on the history of expert selection. To obtain this data, SMARTMOE needs to dump the output of the gating network and synchronize among workers. These operations could be time-consuming if performed too frequently. In practice, we find an effective strategy is only collecting the history of expert selection at a few iterations immediately before the iteration of online adaption. This strategy also utilizes the temporal locality of the expert selection, and prevents collecting useless stale history.

7 Evaluation

7.1 Experimental Setup

Clusters. We evaluate SMARTMOE on three representative clusters, as shown in Table 2, which differ in accelerators, network topology, network bandwidth, and scale. Evaluation on these clusters demonstrates that optimization of SMARTMOE works on different hardware environments.

Table 2: Hardware Platforms for Evaluation.

Name	GPUs Per Node	Max GPUs	Infiniband Bandwidth
blinky	8× NVIDIA V100 PCIe	32	50Gb/s
pinky	4× NVIDIA V100 SXM	64	100Gb/s
inky	8× NVIDIA A100 SXM	32	200Gb/s

Models. The models used for evaluation are shown in Table 3. We choose models from two popular deep learning tasks for evaluation: one is GPT-MoE for natural language processing, and the other is Swin-MoE for computer vision. We use typical batch sizes for each of them. Model parameters are increased along with the number of GPUs. One of the most popular gating methods proposed by GShard [15] is used. GShard gate has a tunable parameter named *capacity factor*, which controls the degree of load imbalance problem (Smaller capacity factor results in a more balanced workload). This optimization is from the model design side to improve MoE model training performance. We apply the GShard gate with different capacity factors to evaluate our system-side optimization.

Table 3: Models Used for Evaluation.

Model	Task	Batch Size	# params (billion)	Capacity Factor
GPT-MoE	NLP	256/512	4.5/7.3/9.9/14.0	$1.2/2.4/4.8/+\infty$
Swin-MoE	CV	4096	0.54/1.0	

Baselines. We compare SmartMoE with four strong training systems. DeepSpeed-MoE [29] is an MoE training system with both system-side and model design-side optimization. It is implemented on DeepSpeed [31] and Megatron-LM [35]. For fairness of comparison, we only turn on the system-side optimization in the following experiments. Tutel [11] targets the scalability problem of MoE training, which achieves promising performance on large-scale MoE training. And Tutel designs Swin-MoE models by extending Swin-Transformer [20, 21] with the MoE technique. FasterMoE [9], the latest version of FastMoE [8], is one of the early efforts of data-sensitive optimization for MoE training. It proposes runtime smart scheduling and expert shadowing. For fairness of comparison, we manually tune hyper-parameters of FasterMoE to achieve good performance. Alpa [41] is a state-of-the-art general-purpose auto-parallelization training system, which hierarchically generates inter- and intra-operator parallel execution plans. It should be noticed that we do not directly compare our system with Alpa. Alpa is implemented on JAX, while the other systems we used are implemented on PyTorch. For fairness of comparison, we use parallelism recommended in the Alpa paper on our system to simulate its performance.

Evaluation Metrics. For end-to-end evaluation, we measure the training latency including forward, backward, gradient synchronization, and optimization stages in both MoE and dense layers in the real model training process. Instead of using randomly generated input samples, we apply real datasets for training to ensure that the dynamic workload in MoE lay-

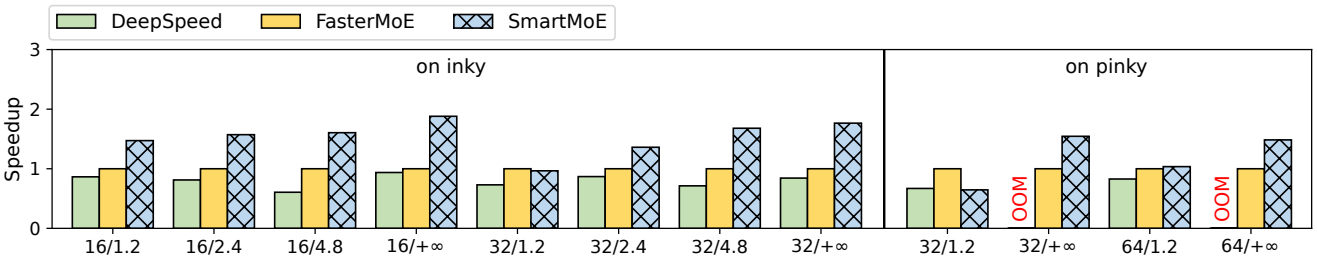


Figure 7: End-to-End Speedup of GPT-MoE Models.

ers represents real situations. For micro-benchmarks, training latency only includes the forward and backward stages in MoE layers. We use a comprehensive expert selection dataset, which is collected in real model training processes, including different model structures and gating methods.

7.2 End-to-End Speedup

We evaluate the end-to-end performance of two types of models on three different clusters. Evaluations are done in a form of weak scaling, where model sizes are increased along with the number of GPUs. And different capacity factors are applied at each scale. X/Y means evaluation on X devices with capacity factor Y. "OOM" means out-of-memory. The performance of FasterMoE is used as a baseline.

Figure 7 shows the end-to-end speedup of GPT-MoE models. SmartMoE achieves on average 1.53× speedup on *inky* cluster, and achieves on average 1.17× speedup on *pinky* cluster. Comparing *inky* cluster and *pinky* cluster, SmartMoE achieves higher speedup on *inky*, where SmartMoE achieves a maximum speedup of 1.88×. This can be explained by two reasons. First, the bandwidth gap between intra-node and inter-nodes links is greater, making hybrid parallelism more efficient. Second, *inky* has more GPUs in a node, increasing possible intra-node parallel strategies. Figure 8 shows the speedup of Swin-MoE models. SmartMoE achieves on average 1.14× speedup on *blinky* cluster.

Comparing different GShard capacity factors, SmartMoE achieves a more significant speedup when the capacity factor is higher. This reveals the tight relationship between model design and system-side optimization. Both of them improve training performance by alleviating the load-imbalance problem in MoE layers, so there are fewer optimization opportunities as the gate is set up with stricter load-balancing limits. In this case, SmartMoE brings improvement because it constructs a better pool. A good pool replaces the originally expensive all-to-all of expert parallelism by communication among fewer workers in hybrid parallelism. Systems with only runtime optimization (e.g. FasterMoE) could reduce the computation overhead by balancing workload, but they fail to reduce the communication overhead, as detailed in §7.5.

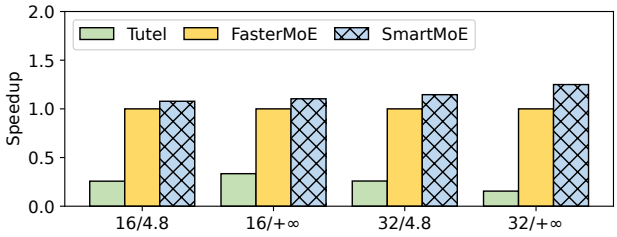


Figure 8: End-to-End Speedup of Swin-MoE Models.

7.3 Offline Parallelization Ablation Study

We study the effectiveness of our offline parallelization algorithm. In order to verify that SmartMoE can find a good pool, we compare performance of different auto-parallelization systems in Figure 9. X/Y means evaluation on X devices with capacity factor Y. The performance of FasterMoE is used as a baseline. To compare with the data-insensitive auto-parallelization approach, we use execution plans recommended by Alpa, in which expert parallelism is used within a node, and pipeline parallelism is used across nodes. To fairly compare the effect of offline parallelization, online optimizations of all systems are disabled. SmartMoE uses a random execution plan generated by the offline data-sensitive auto-parallelization approach. Both data-sensitive and insensitive approaches generate more efficient execution plans compared with pure expert parallelism, attributing to the high performance of hybrid parallelism. Meanwhile, the data-sensitive approach of SmartMoE achieves 2.67× speedup, while the data-insensitive approach only achieves 2.36× speedup.

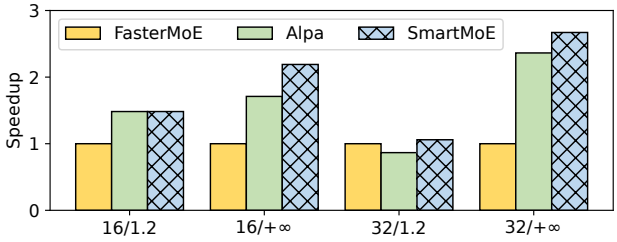


Figure 9: Performance of Offline Parallelization.

Figure 10 shows the accuracy of our data-sensitive performance model. $X-Y$ means evaluation on X devices with local batch size Y. We use MoE layers with expert selection recorded in a real training process to verify the accuracy of SmartMoE performance model. Different scales and batch sizes are used. For all configurations, it achieves $R^2 > 0.5$. Results show that the execution time of an MoE layer varies under different training data. However, the data-insensitive performance model of MoE operators only gives a constant estimation of execution time for each scale and batch size, as shown by vertical lines, inaccurate for most cases. In contrast, our data-sensitive performance model predicts execution time based on current training data. Results show that its accuracy is higher when workers are in the same node, because an unstable cross-node network prevents us from precisely modeling cross-node all-to-all communication latency.

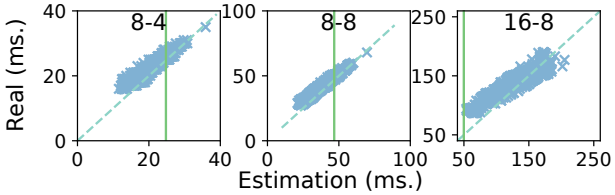


Figure 10: Accuracy of Performance Modeling.

7.4 Online Parallelization Ablation Study

Now we evaluate performance improvement from the adaptive automatic parallelization approach in SmartMoE. A 16-layer MoE model is trained on 64 V100 GPUs of *pinky*. We set the execution plan adjustment frequency to once every 10 iterations. Figure 11 shows the speedup of all 16 MoE layers. SmartMoE achieves on average 1.16× speedup per layer, and at most 1.43× speedup in layer 2. Performance opportunities differ among layers because they are trained to have different internal features. The overhead of execution plan adjustment is insignificant, because training with dynamic execution plans beats the static execution plan at every layer.

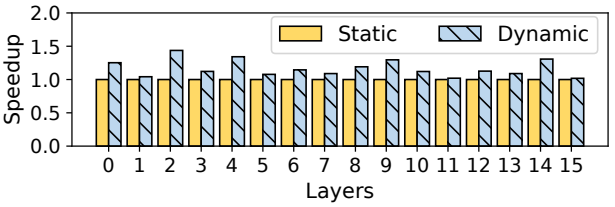


Figure 11: Speedup of Online Parallelization.

In another view, Figure 12 shows the latency of an MoE layer from iteration 1 to 1500. The performance of using either static or dynamic execution plans with different frequencies of adjustment is shown by curves. *dyn.X* denotes switching execution plans every X iteration. Before the first time of switching, the execution plan used is the same as the origin, which does not fit the workload dynamically, resulting in inefficient training. After switching execution plans, it immediately becomes efficient, because SmartMoE uses recent history of expert selections to guide switching. As training progresses, the distribution of expert selection gradually changes. We can find that for frequencies of 250 and 500 iterations, after switching, performance degrades as time increases. This suggests that the frequency of the execution plan adjustment should be high enough for the varying workload. But we also find that the switching overhead could hurt overall performance when adjustment frequency is too high. Moreover, it is interesting that the actual execution plan adjustment tends to be less frequent as the training progresses. For example, in Figure 12, the performance of different switching frequencies becomes more close after iteration 1000. We speculate the reason for this phenomenon is that the distribution of the expert selection becomes more stable after thousands of iterations. In conclusion, we think how to set a proper frequency of dynamic parallelization is still an open problem.

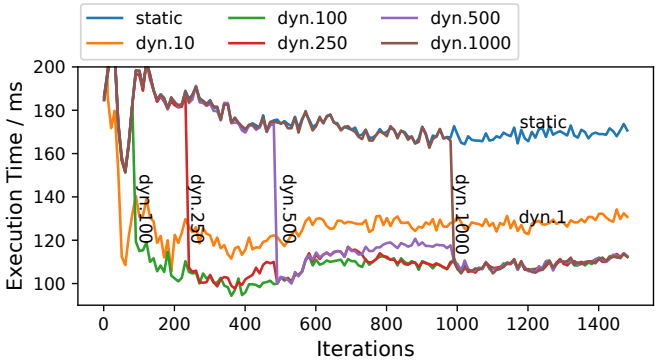


Figure 12: Performance of different adjusting frequencies.

7.5 Fine-Grained Performance Breakdown

We present a fine-grained performance breakdown in Figure 13. FasterMoE is the baseline implementation of MoE model training, which uses pure expert parallelism with simple runtime load-balancing strategies. Alpa represents state-of-the-art training systems with only offline parallelization. SmartMoE represents MoE model training with both offline and online parallelization. Baseline systems and SmartMoE are used to train two MoE models, which are only different in the capacity factor of the gate. The overhead of communication and computation for one iteration is measured separately.

As models with smaller capacity factor tend to have a more balanced workload, the execution time of three systems for the case $capacity = 1.2$ is shorter than its counterpart, $capacity = +\infty$. Because FasterMoE only supports pure expert parallelism, while Alpa and SmartMoE support hybrid parallelism, both communication and computation overhead are reduced by the latter. In the case, $capacity = 1.2$, the speedup of communication is more significant, because com-

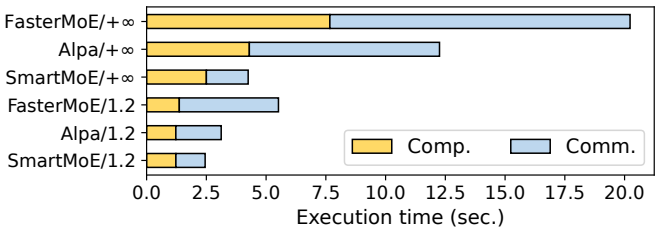


Figure 13: Fine-Grained Performance Breakdown.

putation is forced to be more balanced with the restricted expert selection. SMARTMOE outperforms Alpa because of workload-awareness.

7.6 Overhead Analysis

The overhead of SMARTMOE parallelization approaches is insignificant, compared with the end-to-end training time. Table 4 shows an execution time breakdown of a model which has 16 experts and is trained on a 16 V100 GPUs cluster.

Table 4: Execution Time Breakdown.

Searching	Switching	Forward and Backward		Alpa's Searching
		Original	Optimized	
0.05ms	20ms	75ms	67ms	825s

Searching algorithms cost less than 1ms, because there are only 16 experts. We test the searching algorithms for 1024 experts further, the cost of a single searching process is still less than 50ms. Switching costs 20ms in this example, because the size of expert sub-networks is non-negligible. For a single forward and backward step, the overhead of switching could influence training performance. However, the switching of execution plans brings on average 10% performance gain, which is 8*ms* in this example. After 3 steps, the end-to-end latency of adaptive parallelization is lower than the original one. Typically, tens of forward and backward steps are performed in one iteration, so the switching cost is acceptable for end-to-end training. We also evaluate the overhead of Alpa [41] for this example, which takes 825 seconds to generate an execution plan. The searching overhead of Alpa is orders of magnitude greater than our approaches.

8 Related Work

MoE training systems. Early efforts implement expert parallelism to enable MoE model training in the existing frameworks, including GSPMD [38] for TensorFlow and Fairseq [16], FastMoE [8] for PyTorch. More recent literature has proposed some MoE-specific optimization techniques from different perspectives. To optimize all-to-all communication in MoE training, DeepSpeed-MoE [29] proposes a hierarchical all-to-all algorithm to reduce latency. Lita [17] systematically analyzes all-to-all overhead in MoE training and designs a new communication scheduling scheme. For improving hardware utilization, Tutel [11] delivers adaptive parallelism and pipelining, and scales MoE training to thousands of GPUs. FasterMoE [9] provides a comprehensive performance analysis of MoE training and designs multiple techniques to alleviate load-imbalance problems.

Efforts above focus on the optimization of expert parallelism, which are complementary with SMARTMOE. To enable hybrid parallelism, Tutel [11] combines expert, data, and tensor model parallelism to scale up MoE training. BaGuaLu [23] combines expert and data parallelism to train an MoE model on a full-scale supercomputer. SMARTMOE implements more complete parallelism for MoE models, which brings more performance opportunities.

Automatic parallelization training systems. Previous works target different parallel strategies. Tofu [36] generates tensor model parallel execution plans by a novel DP algorithm. PipeDream [24] and DAPPLE [4] propose pipeline parallelism planners for efficient pipeline partitioning and scheduling. Alpa [41] generates more sophisticated execution plans, considering both inter-operator (i.e., pipeline) and intra-operator (i.e., data and tensor) parallelism.

These efforts are mainly designed for models with dense architecture. SMARTMOE analyzes unique challenges of automatic parallelization for models with sparse architecture, and designs specific techniques to address them.

9 Conclusion

We propose SMARTMOE, an automatic parallelization system for distributed training of sparsely activated models. We identify the key challenge of applying automatic parallelization for sparsely activated MoE models as their dynamic nature or being data-sensitive. To address this challenge, we propose a two-stage solution. The combination space of hybrid parallelism, which enables more potential for optimization, is decomposed by pools. To construct an optimal static pool before training, we design a workload-aware performance model to predict the training performance with estimations of gating networks. At runtime, we invent light-weight searching algorithms to change execution plans with minimum overhead. Compared with selected baselines, SMARTMOE achieves up to 1.88× speedup in end-to-end MoE model training.

10 Acknowledgments

We sincerely thank our shepherd Thaleia Dimitra Doudali and the anonymous reviewers for their valuable feedback on this paper. We also thank Yuyang Jin, Kezhao Huang and Zhenbo Sun for their valuable suggestions. This work is supported by National Key R&D Program of China under Grant 2022ZD0115304 and NSFC for Distinguished Young Scholar (62225206). Jidong Zhai is the corresponding author of this paper (zhaijidong@tsinghua.edu.cn).

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher 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. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[2] 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[3] 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 P. Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen S. 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. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 5547–5569. PMLR, 2022.

[4] 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 *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 431–445. ACM, 2021.

[5] William Fedus, Jeff Dean, and Barret Zoph. A review of sparse expert models in deep learning. *CoRR*, abs/2209.01667, 2022.

[6] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[7] Chaoyang He, Shuai Zheng, Aston Zhang, George Karypis, Trishul Chilimbi, Mahdi Soltanolkotabi, and Salman Avestimehr. SMILE: scaling mixture-of-experts with efficient bi-level routing. *CoRR*, abs/2212.05191, 2022.

[8] Jiaao He, Jiezhong Qiu, Aohan Zeng, Zhilin Yang, Jidong Zhai, and Jie Tang. Fastmoe: A fast mixture-of-expert training system. *CoRR*, abs/2103.13262, 2021.

[9] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 120–134. ACM, 2022.

[10] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[11] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive mixture-of-experts at scale. *CoRR*, abs/2206.03382, 2022.

[12] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Comput.*, 3(1):79–87, 1991.

[13] 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. *CoRR*, abs/2001.08361, 2020.

[14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[15] 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. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[16] Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. BASE layers: Simplifying training of large, sparse models. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 6265–6274. PMLR, 2021.

[17] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Lita: Accelerating distributed training of sparsely activated models. *CoRR*, abs/2210.17223, 2022.

[18] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, 2020.

[19] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 27. ACM, 2021.

[20] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, Furu Wei, and Baining Guo. Swin transformer V2: scaling up capacity and resolution. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 11999–12009. IEEE, 2022.

[21] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021*, pages 9992–10002. IEEE, 2021.

[22] Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H. Chi. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 1930–1939. ACM, 2018.

[23] Zixuan Ma, Jiaao He, Jiezhong Qiu, Huanqi Cao, Yuanwei Wang, Zhenbo Sun, Liyan Zheng, Haojie Wang, Shizhi Tang, Tianyu Zheng, Junyang Lin, Guanyu Feng, Zeqiang Huang, Jie Gao, Aohan Zeng, Jianwei Zhang, Runxin Zhong, Tianhui Shi, Sha Liu, Weimin Zheng, Jie Tang, Hongxia Yang, Xin Liu, Jidong Zhai, and Wenguang Chen. Bagualu: targeting brain scale pretrained models with over 37 million cores. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 192–204. ACM, 2022.

[24] 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 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 1–15. ACM, 2019.

[25] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 7937–7947. PMLR, 2021.

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-lm. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 58. ACM, 2021.

[27] Xiaonan Nie, Pinxue Zhao, Xupeng Miao, Tong Zhao, and Bin Cui. Hetumoe: An efficient trillion-scale mixture-of-expert distributed training system. *CoRR*, abs/2203.14685, 2022.

[28] 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. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[29] 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. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 18332–18346. PMLR, 2022.

[30] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the In-*

ternational Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020, page 20. IEEE/ACM, 2020.

[31] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 3505–3506. ACM, 2020.

[32] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 8583–8595, 2021.

[33] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[34] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[35] 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.

[36] 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, Dresden, Germany, March 25-28, 2019*, pages 26:1–26:17. ACM, 2019.

[37] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake A. Hechtman, and Shibo Wang. Automatic cross-replica sharding of weight update in data-parallel training. *CoRR*, abs/2004.13336, 2020.

[38] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake A. 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. *CoRR*, abs/2105.04663, 2021.

[39] Zhao You, Shulin Feng, Dan Su, and Dong Yu. Speechmoe: Scaling to large acoustic models with dynamic routing mixture of experts. In *Interspeech 2021, 22nd Annual Conference of the International Speech Communication Association, Brno, Czechia, 30 August - 3 September 2021*, pages 2077–2081. ISCA, 2021.

[40] Zhao You, Shulin Feng, Dan Su, and Dong Yu. 3m: Multi-loss, multi-path and multi-level neural networks for speech recognition. *CoRR*, abs/2204.03178, 2022.

[41] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 559–578. USENIX Association, 2022.

MSRL: Distributed Reinforcement Learning with Dataflow Fragments

Huanzhou Zhu*
Imperial College London

Bo Zhao*
Imperial College London and Aalto University

Gang Chen
Huawei Technologies Co., Ltd.

Weifeng Chen
Huawei Technologies Co., Ltd.

Yijie Chen
Huawei Technologies Co., Ltd.

Liang Shi
Huawei Technologies Co., Ltd.

Yaodong Yang
Peking University

Peter Pietzuch
Imperial College London

Lei Chen
Hong Kong University of Science and Technology

Abstract

A wide range of *reinforcement learning* (RL) algorithms have been proposed, in which agents learn from interactions with a simulated environment. Executing such RL training loops is computationally expensive, but current RL systems fail to support the training loops of different RL algorithms efficiently on GPU clusters: they either hard-code algorithm-specific strategies for parallelization and distribution; or they accelerate only parts of the computation on GPUs (e.g., DNN policy updates). We observe that current systems lack an abstraction that decouples the *definition* of an RL algorithm from its *strategy* for distributed execution.

We describe MSRL, a distributed RL training system that uses the new abstraction of a *fragmented dataflow graph* (FDG) to execute RL algorithms in a flexible way. An FDG is a heterogeneous dataflow representation of an RL algorithm, which maps functions from the RL training loop to independent parallel dataflow *fragments*. Fragments account for the diverse nature of RL algorithms: each fragment can execute on a different device using its own low-level dataflow implementation, e.g., an operator graph of a DNN engine, a CUDA GPU kernel, or a multi-threaded CPU process. At deployment time, a *distribution policy* governs how fragments are mapped to devices, without changes to the algorithm implementation. Our experiments show that MSRL exposes trade-offs between different execution strategies, while surpassing the performance of existing RL systems.

1 Introduction

Reinforcement learning (RL) solves decision-making problems by having agents learn policies – typically represented as deep neural networks (DNNs) – on how to act in an environment [51]. RL has achieved remarkable outcomes: in game play, AlphaGo [49] defeated a world champion in the Go board game; in biology, AlphaFold [21] predicts three-dimensional structures for protein folding; in robotics, RL allows robots to perform dexterous manipulation without human intervention [15]; and the ChatGPT chatbot [41] uses a reinforcement step with PPO [47] to fine-tune its model.

Such advances in RL, however, come with high computational demands: AlphaStar trained 12 agents on 384 TPUs and 1,800 CPUs for 44 days to achieve grandmaster level in StarCraft II game play [54]; OpenAI Five trained to play Dota 2 games for 10 months with 1,536 GPUs and 172,800 CPUs [3].

Existing RL systems (e.g., SEED RL [8], Acme [18], Ray [34], RLlib [25], Podracer [16]) are therefore optimized for specific types of RL algorithms and the structure of their RL training loops. In particular, systems hardcode a strategy for parallelizing and distributing the RL computation:

Parallelization. Most RL systems only accelerate the DNN computation on GPUs or TPUs [8, 18, 25] using current DNN engines (e.g., PyTorch [42], TensorFlow [14], and MindSpore [19]). Other parts of RL algorithms (e.g., action generation, environment execution, and trajectory sampling) are executed as sequential Python functions on worker nodes, potentially becoming performance bottlenecks.

Some systems try to accelerate more parts of RL training loops: Podracer [16] uses the JAX [12] compilation framework to vectorize Python implementations of RL algorithms; WarpDrive [23] implement the entire RL training loop using CUDA on a GPU; and RLlib Flow [26] uses a set of parallel dataflow operators [58] to express an RL training loop. All of these approaches, however, require users to rewrite the *complete* RL algorithm (e.g., agents, learners, and environments) using a single API with a fixed set of dataflow operators.

Distribution. When distributing computation, current RL systems allocate algorithmic components (e.g., actors and learners) to workers in a fixed way: SEED RL [8] assumes that learners perform policy inference and training on TPUs, and actors execute on CPUs; Acme [18] only distributes actors and maintains a single learner; and TLeague [50] distributes learners but co-locates environments with actors on CPU workers. As we shown in §6, such decisions are algorithm-specific: since different algorithms deployed on a given set of resources exhibit diverse bottlenecks, a single distribution strategy cannot exhibit the best performance in all cases.

*Equal contribution.

We observe that the above challenges come from a lack of separation between the *definition* of an RL algorithm and how it is *executed* by the system. For example, many RL systems allow users to define RL algorithms as a set of Python functions for agents, learners, and environments. The system then directly invokes the implementation of e.g., an agent's `act()` function to produce new actions for the environment. While this simplifies system implementation, it removes control from the system regarding how algorithmic components are parallelized or distributed at deployment time.

DNN training systems use intermediate representations (IRs), which are compiled to target devices for execution, to decouple DNN definition from execution [1,6,56]. Such approaches, however, assume a homogeneous training computation (forward/backpropagation over differentiable DNNs [2]), which can be expressed by a fixed set of computational operators over tensor types. In contrast, the space of RL algorithms exhibits more heterogeneity in terms of the computation performed by algorithmic components (agents, actors, learners, polices, environments, leaderboards), their exchanged data (observations, actions, policy updates) and communication patterns (one-to-one, one-to-many, all-reduce).

Our goal is to explore a new design for an RL training system that requires users to define an RL algorithm only once. At deployment, the system then supports (i) the execution of arbitrary parts of the RL computation on parallel devices (GPUs and CPUs); and (ii) the deployment of parts of the computation on distributed workers.

We describe **MSRL**, a distributed RL system that achieves this by decoupling the specification of a RL algorithm from its execution through the abstraction of a *fragmented dataflow graph* (FDG). Unlike dataflow approaches of DNN and data analytics systems, an FDG does not enforce a single uniform dataflow representation, which is challenging for diverse RL algorithms. Instead, it allows different components of an RL algorithm to have bespoke GPU or CPU implementations, chosen by the user at deployment time.

In summary, MSRL's design makes three contributions:

(1) Fragmented dataflow graphs (§3). From the RL algorithm implementation, MSRL constructs an FDG, which consists of independent *fragments*. Each fragment can have its own dataflow representation (e.g., DNN operators, CUDA, or Python) targeting GPUs or CPUs. MSRL then maps instances of fragments to devices at deployment time.

To obtain fragments, MSRL statically analyzes the RL algorithm implementation to group functions into fragments. By default, the boundaries between fragments are chosen based on the algorithmic components of the RL algorithm. Since fragments are deployed on different devices, MSRL synthesizes appropriate communication operators that allows fragments to exchange data.

(2) API with distribution policies (§4). Users specify an RL algorithm by implementing its algorithmic components as Python functions in a traditional way. The implementation makes no assumptions about how the algorithm will be executed: all runtime interactions between components are managed by calls to MSRL APIs. A separate *deployment configuration* defines the devices available for execution.

Since FDGs separate algorithm implementations from execution, MSRL can apply different *distribution policies* to govern how fragments are mapped to devices. MSRL supports distribution policies, which subsume the hard-coded distribution strategies of current RL systems: e.g., a policy can distribute multiple actors to scale environment interaction (like Acme [18]); distribute actors and move policy inference to learners (like SEED RL [8]); distribute both actors and learners (like Sebulba [16]); or represent the full RL training loop on a GPU (like WarpDrive [23] and Anakin [16]).

When a user changes the algorithm configuration, its hyper-parameters or deployment resources, they can also switch between distribution policy to maintain high training efficiency without having to change the RL algorithm implementation.

(3) Heterogeneous fragment execution (§5). For execution, MSRL deploys hardware-specific implementations of fragments on CPUs and GPUs. MSRL supports different fragment implementations: CPU implementations use regular (multi-process) Python code, and GPU implementations are generated as compiled computational graphs for DNN engines (e.g., MindSpore or TensorFlow) if a fragment is implemented using operators, or are implemented directly in CUDA.

MSRL optimizes co-located fragments on the same worker: it fuses data-parallel fragments for more efficient execution by batching data items (e.g., tensors) and using single-instruction-multiple-data (SIMD) execution.

We evaluate MSRL experimentally and show that MSRL's abstraction supports flexible training across different RL algorithm without compromising training performance compared to current hardcoded RL training systems: MSRL scales to 64 GPUs and outperforms the Ray distributed RL system [34] by up to 3×. By switching between distribution policies, MSRL can improve the training time of the PPO RL algorithm by up to 2.4× as hyper-parameters, network properties or hardware resources change.

2 Distributed Reinforcement Learning

Next we give background on RL algorithms (§2.1), discuss the requirements for RL training (§2.2), and survey the design space of existing RL systems (§2.3).

2.1 Reinforcement learning

Reinforcement learning (RL) solves a sequential decision-making problem in which an *agent* operates in an *environment*. The agent's goal is to learn a *policy* that maximizes the cumulative *reward* based on the feedback from the environment (see Fig. 1). RL training performs three steps: ❶ *policy inference*: an agent obtains an action by performing inference on a policy; ❷ *environment execution*: the environment executes the action, generating *trajectories* of $\langle state, reward \rangle$

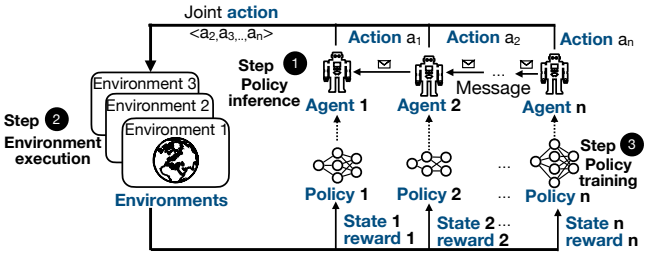


Fig. 1: RL training loop with multiple agents

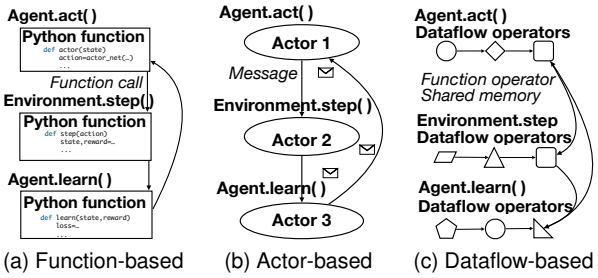


Fig. 2: Types of RL system designs

pairs; and ❸ *policy training*: the agent improves the policy by adapting it based on the reward.

RL algorithms are diverse in nature, falling into three categories: (1) *value-based* algorithms (e.g., DQN [33]) use a DNN to approximate a value function that predicts the expected return of actions. Agents then select actions based on these estimated values; (2) *policy-based* algorithms (e.g., Reinforce [55]) directly learn a parameterized policy – approximated by a DNN – for selecting actions without a value function. Agents use batched trajectories to train the policy by updating its parameters to maximize the reward; and (3) *actor–critic* algorithms (e.g., PPO [47], DDPG [27], A2C [32]) combine the two by learning a policy that selects actions and a value function that evaluates them.

Multi-agent reinforcement learning (MARL) employs multiple agents, each optimizing its own cumulative reward when interacting with the environment or other agents (see Fig. 1). A3C [32] executes agents asynchronously on separate environment instances; MAPPO [57] extends PPO to a multi-agent setting in which agents share a global parameterized policy.

2.2 Requirements for distributed RL systems

RL algorithms explore large spaces of actions, states and DNN parameters, which grow exponentially with the number of agents [37]. RL systems must thus exploit the parallelism of GPUs and scale computation to many worker nodes.

Due to the diverse computational patterns exhibited by different RL algorithms, there is no single strategy for parallelization and distribution that is optimal for all RL algorithms, e.g., in terms of both achieving the lowest iteration time and scaling to the most workers. Bottlenecks during training depend on the specific algorithm, the training workloads and the employed hardware resources: e.g., our experiments show that, for PPO [47], environment execution (❷) takes up to 98% of execution time; for MuZero [46], a large MARL algorithm with many agents, instead 97% of time is spent on policy inference and training (❶+❸).

Therefore, there are many proposals how to parallelize and distribute RL computation: e.g., in single-agent RL, environment execution (❷ in Fig. 1), policy inference and training (❶+❸) can be distributed across workers [8, 16, 16, 18, 34]; in MARL, agents can be distributed [25, 26, 34, 45] and exchange training state [29, 39]. Environment instances can execute in parallel [16, 32] or be distributed [7].

Instead of committing to one approach for parallelizing and distributing the RL computation, an RL system should provide the flexibility to change its execution approach based on the workload. This leads us to the following requirements:

(1) Execution abstraction. The system should have a flexible execution abstraction for parallelizing and distributing computation, unencumbered by how the algorithm is defined.

(2) Distribution strategies. The system should support multiple strategies for distributing RL computation. Users should be permitted to switch between strategies based on the training workload, without changes to the algorithm.

(3) Acceleration support. The system should exploit the parallelism of GPUs and CPUs, accelerating not just policy training and inference (❶+❸) but the full RL training loop, including environment execution (❷) [23].

(4) Algorithm abstraction. The system should expose familiar APIs to users for defining RL algorithms and their training loops, structured around algorithmic components [9, 13, 22], such as agents, actors, learners, policies, environments, etc.

2.3 Design space of existing RL systems

We analyze the design space of RL systems. Existing designs fit into three types (see Fig. 2):

(a) Function-based RL systems are the most common type. They express RL algorithms typically as Python functions, executed directly by workers (see Fig. 2a). The RL training loop is implemented through direct function calls. For example, Acme [18] and SEED RL [8] organize algorithms as actor/learner functions; RLGraph [45] uses a component abstraction, and users register Python callbacks to define functionality. Distributed execution is delegated to backend engines, e.g., TensorFlow [14], PyTorch [42], Ray [34].

(b) Actor-based RL systems execute algorithms as a set of (programming language) actors deployed on worker nodes (see Fig. 2b). For example, Ray [34] uses an actor model to define tasks, which are distributed among nodes using remote calls. Defining control flow in an actor model, however, is burdensome. To overcome this issue, RLlib [25] adds logically centralized control on top of Ray. Similarly, MALib [61] offers a higher level abstractions for population-based MARL algorithms (e.g., PSRO [35]) on Ray.

(c) Dataflow-based RL systems define algorithms through a

Type	System	(1) Execution	(2) Distribution	(3) Acceleration	(4) Algorithm
Function-based	SEED RL [8]	Python functions	environment only	DNNs only	actor/learner/env
	Acme [18]	Python classes			
	RLGraph [45]		delegated to backend		agent
Actor-based	Ray [34]	task (stateless) actor (stateful)	scheduler/RPC	DNNs only	Python functions with Ray API [34]
	RLlib [25]				
	MALib [61]				agent/actor/learner/env
Dataflow-based	Podracer [16]	JIT-compiled by JAX [12]	hardcoded	funcs/DNNs/envs	JAX [12] API
	RLlib Flow [26]	predefined dataflow operators	dataflow operators/ Ray tasks [34]	DNNs only	operator API
	WarpDrive [23]	GPU thread blocks	—	CUDA kernels	CUDA API
Fragmented dataflow	**MSRL**	heterogeneous fragments	any fragment	funcs/operators/ DNNs/envs	agent/actor/learner/env

Tab. 1: Design space of distributed RL systems

set of data-parallel operators, implemented by GPU kernels or distributed tasks (see Fig. 2c). Users must express the complete RL training loop using operators APIs. For example, Podracer [16] uses JAX [12] to compile vectorized Python to TPU kernels. RLlib Flow [26] provides Spark-like dataflow operators on top of Ray. WarpDrive [23] executes RL training loops implemented in CUDA using GPU thread blocks.

Tab. 1 considers how well these approaches satisfy the four requirements from §2.2:

(1) Execution abstraction. Function- and actor-based systems execute RL algorithms directly through implemented (Python) functions and user-defined language actors, respectively. This prevents systems from applying optimizations how RL algorithms are parallelized or distributed. In contrast, dataflow-based systems execute computation using operators [16, 26] or CUDA kernels [23]. This allows for execution optimizations, but algorithm implementations are restricted by the supported set of operators.

(2) Distribution strategies. Most function-based systems only support a hardcoded strategy, e.g., one that distributes actors to parallelize the environment execution (❶+❷ in Fig. 1) with a single learner. In actor-based systems, a scheduler assigns stateful actors and stateless tasks to workers, and users have no control over the distribution approach.

Similarly, existing dataflow-based systems only support fixed policies how dataflow operators are assigned to workers: *Anakin* [16] co-hosts an environment and an agent on each TPU core; *Sebulba* distributes the environment, learners and actors on different TPUs; and RLlib Flow [26] shards dataflow operators across distributed Ray actors.

(3) Acceleration support. Most RL systems only accelerate DNN policy inference and training (❶+❸). Some dataflow-based systems (e.g., Podracer [16] and WarpDrive [23]) also accelerate other parts of training, requiring bespoke dataflow implementations: e.g., Podracer accelerates environment execution (❷) on TPU cores; WarpDrive executes the entire RL training loop (❶–❸) on a single GPU using CUDA.

(4) Algorithm abstraction. Function-based RL systems provide intuitive actor/learner/env APIs. Actor-based RL systems exhibit harder-to-use low-level APIs for distributed components (e.g., Ray's `get`/`wait`/`remote` [34]) and must rely on high-level libraries (e.g., RLlib's `PolicyOptimizer` API [25]) to bridge the gap. Dataflow-based systems come with their own dataflow operators, requiring the rewriting of a complete RL training loop. For example, JAX [12] require users to express RL algorithms in terms of the `vmap` and `pmpa` operators for vectorization and single-program multiple-data (SPMD) parallelism, respectively.

We note that there is an opportunity to combine the usability of a function-based algorithmic abstraction, which allow users to express RL algorithms naturally using algorithmic components, with the acceleration potential of dataflow-based approaches. Such a design, however, requires a new execution abstraction, which also retains the flexibility of supporting different distribution strategies.

3 Fragmented Dataflow Graphs

We now describe the dataflow abstraction that we use to represent the heterogenous computation of RL algorithms and to map it to various devices for execution.

3.1 Overview

Our aim is to take an arbitrary RL training loop of a single- or multi-agent RL algorithm (Fig. 1) and translate it to a dataflow representation. The RL system can then use the dataflow representation to parallelize and distribute the computation across heterogeneous devices. We observe that RL training loops combine different types of computations: e.g., actors decide on an action to carry out based on inference results from the DNN policy, obtaining trajectories first; learners update the DNN policy using a DNN training algorithm; and environments execute steps in e.g., a physics simulator, returning trajectories based on the current simulation state.

Unlike existing dataflow models for DNN computation [1, 6, 19], this heterogeneity of computation makes it challenging to impose a single uniform dataflow model that prescribes a set of computational operators and a single data representation (e.g., tensors) between them. Instead, we adopt a *heteroge-*

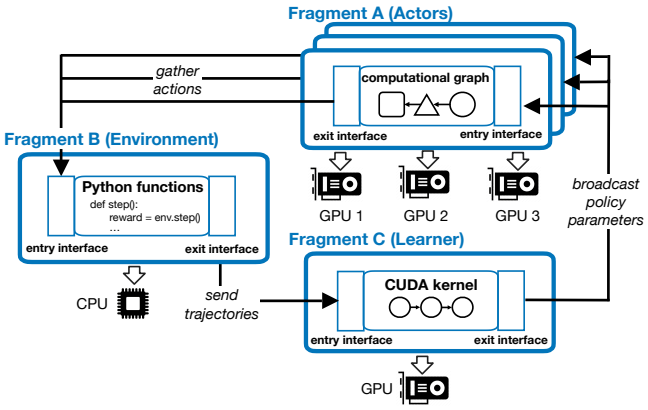


Fig. 3: Fragmented dataflow graph

neous dataflow model, in which independent dataflow representations for different algorithmic components of the RL training loop can be "stitched together" through well-defined interfaces. We refer to this dataflow model as a *fragmented dataflow graph* (FDG), shown in Fig. 3.

Fragments. Each node in an FDG is a potentially data-parallel *fragment*, which is implemented using a bespoke dataflow representation. For example, fragment A in Fig. 3 represents the action computation of an *actor* using the data-parallel operators of a DNN engine [1, 6, 19], performing model inference to decide on an action; fragment B implements the environment simulation directly as parallel Python code; and fragment C conducts the model training, which is implemented as CUDA kernels [11, 38].

Based on the fragment allocation, FDGs support the execution of RL computation on different devices. Each fragment is assigned to one or more devices: the DNN operator representation of fragment A allows it to be deployed on GPUs or CPUs; fragment B requires a Python interpreter with the `multiprocessing` library [10] on CPU cores; and instances of fragment C must run as CUDA kernels on GPUs.

In addition, it is possible to parallelize fragment execution by having multiple instances of a fragment and assigning each instance to a separate device. In Fig. 3, fragment A is replicated 3 times and executed by 3 GPU devices in parallel.

Communication. To form a connected FDG of the complete RL training loop, each fragment must support *entry* and *exit* interfaces, allowing them to exchange data: the entry interface receives data as a byte buffer, which is transformed into a fragment-specific representation (e.g., a tensor); and the exit interface requires a fragment to provide output, which is serialized for consumption by the next fragment.

The implementation of these interfaces depends on how the fragments are deployed: if two fragments are placed on different workers, the interface must use network communication to exchange data e.g., using an RPC protocol over Infiniband [48]; if two fragments are co-located on devices on the same worker, they can share data structures e.g., using inter-GPU communication links such as NVLink [40].

According to the communication method and distribution policy (§4.2), fragment interfaces may be *blocking*, which means that they only execute after all data has arrived, e.g., after a collective communication `AllReduce` round when aggregating DNN gradients. Alternatively, they can be *non-blocking*, which means that they execute continuously, e.g., allowing actors to interact with environments asynchronously.

3.2 Trade-offs with fragmented dataflow graphs

FDGs subsume execution strategies of existing RL systems. For example, an FDG may represent an actor and its environment as a single CPU-based fragment, and a learner as a GPU-based fragment, as proposed by Acme [18]. Alternatively, it may create a larger GPU fragment by moving the DNN policy to the learner, accelerating policy inference, as proposed by SEED RL [8]. An even larger fragment may contain the actor, learner, policy, and environment, executing the whole training loop on a single GPU, as proposed by WarpDrive [23] and Anakin [16].

More generally, FDGs expose two dimensions that impact execution performance:

Fragment granularity refers to the code size, which affects device utilization: a small fragment may underutilize a GPU, and a large one may exhaust GPU memory.

Fragment granularity also determines the ratio between computation and communication. The frequency and amount of data synchronization between fragments often limit scalability: coarser fragments require less synchronization with other fragments, which reduces communication overhead, but they remove opportunities for parallelism. For example, multiple fragments may exchange trajectories frequently at each step; alternatively, they may batch data from multiple steps and communicate only once in each episode.

Fragment co-location is the assignment of fragments to devices on the same worker. Co-locating fragments avoids network communication (e.g., Ethernet or InfiniBand) and instead uses more efficient intra-node communication (e.g., NVLink or PCIe). Whether two fragments can be co-located depends on the available resources on the worker, such as the number of available GPUs.

Choosing the right trade-off between fragment granularity and co-location is key to achieving good performance. In the next section, we describe how FDGs allow users to define an RL algorithm and select between different distribution policies, which expose these trade-offs.

4 Using MSRL

MSRL is our system that implements FDGs for parallel and distributed execution of RL algorithms based on distribution policies. We describe the APIs supported by MSRL for users to define RL algorithms (§4.1) and the distribution policies supported by MSRL to deploy FDGs (§4.2).

Type	API	Description
Component	`Agent, Actor, Learner, Trainer`	Abstract classes for components
	`Actor.act(...)`	Trajectory collection
	`Learner.learn(...)`	DNN policy training
	`Trainer.train(...)`	RL training loop
	`MSRL.agent_act(...)`	Invoke actor
	`MSRL.agent_learn(...)`	Invoke learner
	`MSRL.env_step(...)`	Execute environment
	`MSRL.env_reset()`	Reset environment
Interaction	`MSRL.replay_buffer_insert(...)`	Store trajectories in buffer
	`MSRL.replay_buffer_sample()`	Sample trajectories from buffer

Tab. 2: MSRL APIs

4.1 MSRL APIs

MSRL's APIs are designed to decouple the algorithm logic from its deployment, while supporting familiar algorithmic concepts (i.e., agents, actors, learners, trainers, and environments). As listed in Tab. 2, MSRL supports *component* and *interaction* APIs:

Component APIs specify an RL algorithm by defining algorithmic components derived from abstract classes. An `Agent` consists of actors and learners: actors collect trajectories in `Actor.act()` by invoking `MSRL.env_step()`; and learners implement the DNN update logic in `Learner.learn()`. A *trainer* constructs the RL training loop in `Trainer.train()`. It can use `MSRL.env_step()` to invoke the environment implementation and `MSRL.env_reset()` to reset the training episode.

Interaction APIs offer RL-specific functionality to algorithmic components. For example, an *actor* can store collected trajectories in a replay buffer using `MSRL.replay_buffer_insert()`, and a *learner* can sample from that replay buffer with `MSRL.replay_buffer_sample()`. This avoids direct invocations between components, which allows MSRL to distributed fragments transparently.

Alg. 1 shows a sample implementation of the multi-agent PPO (MAPPO) algorithm [57]. (For brevity, it omits the DNN policy definition.) The `MAPPOAgent` (line 1) defines the agent behavior: it interacts with the environment through `MAPPOActor` (line 6), and performs the policy training with `MAPPOLearner` (line 12). The agent collects trajectories (lines 8–9), and updates its DNN policy (lines 15–21).

`MAPPOTrainer` defines the RL training loop (line 23). At the start of each episode, it resets the environment (`MSRL.env_reset()`) and calls `MSRL.agent_act()` to place trajectories (line 28) in a replay buffer (line 10). The trainer invokes the learner through `MSRL.agent_learn()` (line 29).

To separate the algorithm's logic from its deployment, MSRL uses configurations, specified as Python dictionaries: an *algorithm configuration* instantiates the algorithmic components and their hyper-parameters (e.g., the number of agents and learning rates). In the MAPPO example (lines 30–38), the configuration requests 4 agents, each with 3 actor and 1 learner. Each actor interacts with 32 environments; and a *deployment configuration* defines (i) the resources (e.g., GPUs, CPUs, and worker nodes) and (ii) a *distribution policy*. In the example (lines 39–42), it deploys workers with 4 GPUs each, using the SingleLearnerCoarse distribution policy.

Algorithm 1: MAPPO algorithm in MSRL

```
1  class MAPPOAgent(Agent):
2    def act(self,state):
3      return self.actors.act(state)
4    def learn(self,sample):
5      return self.learner.learn(sample)
6  class MAPPOActor(Actor)
7    def act(state):
8      action = self.actor_net(state)
9      reward,new_state = MSRL.env_step(action)
10     MSRL.replay_buffer_insert(reward,new_state)
11     return reward,new_state
12 class MAPPOLearner(Learner):
13   def learn():
14     sample = MSRL.replay_buffer_sample()
15     action,reward,state,next_state = sample
16     last_pred = self.critic_net(next_state)
17     pred = self.critic_net(state)
18     r = discounted_reward(reward,last_pred,self.gamma)
19     adv = gae(reward,next_state,pred,last_pred,self.
           gamma)
20     for i in range(self.iter):
21       loss += self.mappo_net_train(action,state,adv,r)
22     return loss / self.iter
23 class MAPPOTrainer(Trainer):
24   def train(self,episode):
25     for i in range(episode):
26       state = MSRL.env_reset()
27       for j in range(self.duration):
28         reward,new_state = MSRL.agent_act(state)
29       loss = MSRL.agent_learn()
30 mappo_algorithm_config = {
31   'agent':{'num':4,'name':MAPPOAgent,
32            'actor':MAPPOActor,'learner':MAPPOLearner},
33   'actor':{'num':3,'name':MAPPOActor,
34            'policy':MAPPOActorNet,'env':True},
35   'learner':{'num':1,'name':MAPPOLearner,
36              'policy':[MAPPOCriticNet,MAPPONetTrain],
37              'params':{'gamma':0.9}},
38   'env':{'name':MPE,'num':32,'params':{'name':'MPE'}}}
39 mappo_deployment_config = {
40   'workers':[198.168.152.19, 198.168.152.20, [...],
41   'GPUs_per_worker':4},
42   'distribution_policy':'SingleLearnerCoarse'}
```

4.2 Distribution policies

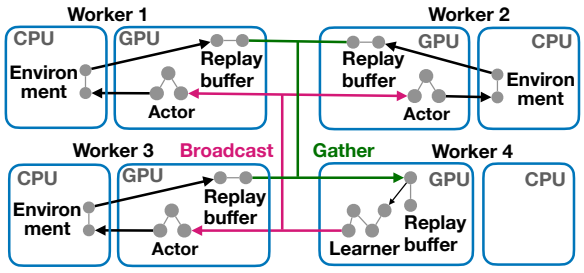
A *distribution policy* (DP) governs how MSRL distributes and parallelizes an RL algorithm by allocating, replicating and collocating fragments from the FDG to workers and devices.

In general, there exists no single DP that is optimal in all cases: the performance and applicability of a DP depends on the type of RL algorithm, the size and complexity of the DNN model, its hyper-parameters, the available cluster compute resources (i.e., CPUs and GPUs), and the network bandwidth. MSRL allows users to easily switch between DPs, either for the same RL algorithm or when using different algorithms. MSRL provides six DPs, which follow widely used hard-coded distribution strategies of existing RL systems.

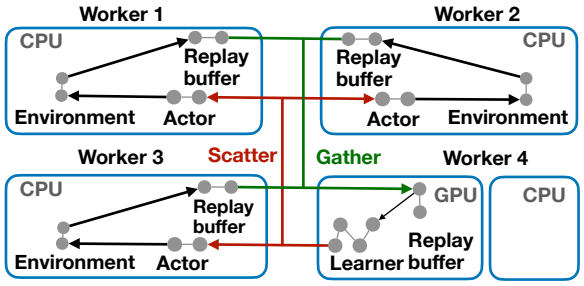
Next we give an overview of the support DPs and their trade-offs. Tab. 3 shows how three of the DPs deploy the fragments of an RL algorithm. (We list all DPs currently implemented by MSRL in Appendix A).

DP-SingleLearnerCoarse replicates the actor and environment fragments but uses a single learner. The policy DNN is replicated across the actors and learner, which only requires coarse synchronization. This policy is therefore most suitable with

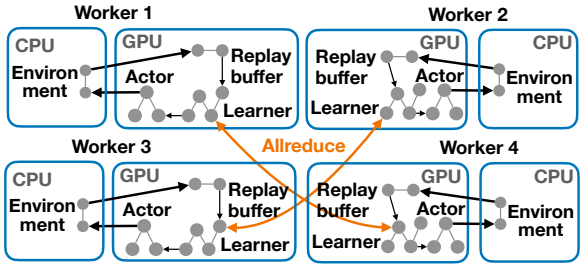
DP-SingleLearnerCoarse
replicate: *actor*,*env* split: *learner* e.g., Acme [18], Sebulba [16]



DP-SingleLearnerFine
replicate: fused *actor/env* split: *learner* e.g., SEED RL [8]



DP-MultiLearner
replicate: fused *actor/learner*, *env*



Tab. 3: Sample distribution policies with deployments

computationally-expensive environments that need scaling out, but small DNN models that can be synchronized in a batched fashion, e.g., Acme [18], Sebulba [16].

The MAPPO deployment in Alg. 1 uses DP-SingleLearnerCoarse: each agent is partitioned into 4 GPU fragments, i.e., 3 actors and 1 learner, and 3 CPU fragments for environments. Actor and environment fragments are collocated. This setting is replicated for each of the 4 MAPPO agents, as specified in the algorithm configuration.

In contrast, DP-SingleLearnerFine fuses the actor and environment into a single CPU fragment, and only deploys the learner on a GPU. Therefore it does not communicate policy parameters between workers, which is preferable for large DNN models with many parameters. Compared to the DP-SingleLearnerCoarse, it relies on fine-grained synchronization: training data is exchanged at each step, instead of being batched per episode. For good performance, DP-SingeLearnerFine therefore requires high bandwidth connectivity between workers, e.g., SEED RL [8].

DP-MultiLearner performs data-parallel training with multiple learners. This policy is necessary when the data generated from actors becomes too large for a single GPU, and e.g., DP-SingleLearnerCoarse cannot be used. However, it requires the tuning of hyper-parameters (e.g., the learning rate) to scale due to its reliance on data parallelism. Since workers only exchange information about the trained policy (e.g., aggregated DNN gradients), DP-MultiLearner is communication efficient, supporting fully decentralized MARL training [5, 43, 59, 62].

MSRL supports further policies: DP-GPUOnly fuses the RL training loop into a single GPU fragment and distributes it to multiple GPU devices. DP-Environments dedicates one or more workers for the execution of complex or compute-intensive environments (e.g., physics simulations). Finally, DP-Central introduces a separate fragment for a centralized component (e.g., policy pool [61] or parameter server [24]).

The choice of the best distribution policy depends on the algorithm's characteristics and available hardware resources: single-agent RL algorithms, such as PPO/A3C, exhibit the best performance under the DP-SingleLearnerCoarse policy, which distributes actors to speed up trajectory collection through data parallelism; multi-agent algorithms, such as MAPPO/MADDPG, require a DP-MultiLearner policy that distributes actors and learners separately from agents, thus parallelizing both trajectory collection and model training; a DP-GPUOnly policy can be used in a GPU environment to fuse the training loop and execute it entirely on GPUs, which offers the best performance.

Based on the hardware resources, bottlenecks shift between DPs: the DP-SingleLearnerFine policy exchanges data between actors/environments at a fine granularity by distributing inference/training to one GPU worker and environments across CPU workers. Despite the need for frequent communication, this policy is suitable in situations where GPUs are scarce; in contrast, the DP-SingleLearnerCoarse policy co-locates the GPU DNN inference with the environments, enabling the learner to gather batched training data. With enough GPUs, this policy accelerates trajectory collection.

5 MSRL Architecture

We describe MSRL's architecture, explaining how FDGs are generated (§5.1) and executed (§5.2).

MSRL follows a coordinator/worker design (see Fig. 4): a user submits the RL algorithm implementation to the *coordinator* ❶. The coordinator generates the fragments that constitute the FDG and dispatches them to the workers ❷. Each worker maintains one or more *execution backends* (e.g., a DNN engine, a CUDA job scheduler, a Python interpreter), each managing devices (e.g., GPUs or CPU cores). After receiving fragments, the worker optimizes them ❸ and submits them to a backend for execution ❹.

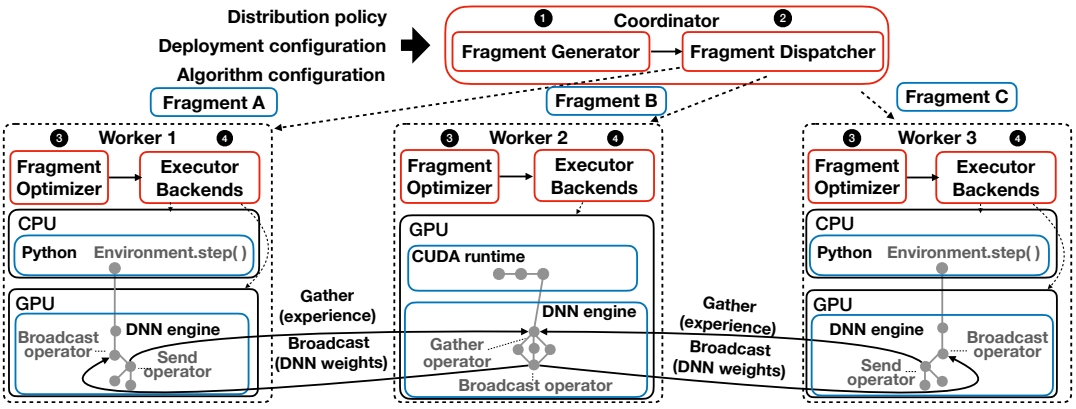


Fig. 4: Overview of the MSRL architecture

5.1 Generating FDGs

The coordinator has two components:

The **FDG Generator** partitions the RL algorithm according to the DP specified in the deployment configuration (§4.1). It splits the implementation at fragment boundaries and injects code for the interface implementations between fragments. The fragment logic is then emitted as part of a `run()` method in a generated `Fragment` class.

The partitioning of the RL algorithm into fragments uses the information associated with a DP. Each DP provides a set of rules about (1) how fragments are generated and (2) how they are distributed. The DP contains a *fragment template*, which associates each fragment with a Python class that has placeholders for class names, member functions, and other relevant elements. These placeholders instruct the Generator where to insert specific algorithm logic, such as actor computation, into the generated fragments. The DP also defines the communication operations required by the interfaces. To choose appropriate implementations, the DP refers to communication operations supported by backends (e.g., `comms.AllGather` [20] in a DNN engine). The DP also specifies which fragments are replicated into multiple instances for parallel execution, or co-located on the same worker.

When partitioning the RL training loop, the boundaries between fragments follow the algorithmic components (actors, learners, environments). The data to be transferred between fragments is defined in terms of the function signatures of the components. The partitioning is done on a dataflow representation of the RL algorithm: nodes in the dataflow graph are Python statements; edges represent the dataflow through variables. Therefore, edges at the boundary of algorithmic components describe fragment interfaces, and we refer to them as *boundary edges*. MSRL creates fragments by partitioning the dataflow graph at these boundary edges.

As an example, consider partitioning the MAPPO algorithm (Alg. 1) into actor and learner fragments, with the boundary between lines 28 and 29. Fig. 5a shows the simplified dataflow graph obtained after static analysis, with the input/output data of the components shown in red. Splitting the graph at these boundaries, partitions it into two fragments (see Fig. 5b), which communicate through the new interface obtained from the boundary edges (shown in red).

Algorithm 2 Generation of fragmented dataflow graphs

function generate_FDG (*alg*, *DP*):
1: *FDG* ← {}, *DFG* ← *generate_DFG(alg)*
2: *boundary_edges* ← *obtain_boundary_edges(DFG)*
3: *interfaces* ← *generate_interfaces(boundary_edges, DP)*
4: **for** *boundary* in *boundary_edges* **do**
5: *fragment_code* ← *build_fragment(alg, boundary)*
6: *fragment* ← *build_fragment(fragment_code, interfaces, DP)*
7: *FDG* ← *FDG* ∪ *fragment*
8: **return** FDG

Alg. 2 summarizes the FDG generation. The Generator takes the RL algorithm's abstract syntax tree (*alg*) and distribution policy *DP* as input (line 0) and constructs its dataflow graph (*DFG*) (line 1). Next, it locates the algorithmic components and determines the boundary edges from the DFG (line 2). Based on the information from the DP, it constructs the communication interfaces (line 3). For each boundary edge (line 4), it extracts the fragment code (line 5) and builds the fragment with its interface implementation (line 6). At the end, it returns the complete FDG (line 8).

The **Fragment Dispatcher** launches instances of execution backends on each worker according to the devices from the deployment configuration. It also sets up distributed communication, e.g., through MPI [30], as required by the fragment interfaces. Finally, it assigns fragments to devices based on the DP and sends the fragments to the workers.

5.2 Executing fragments

The workers use a set of *execution backends* to take the fragment code and run it. Some backends (e.g., a DNN engine) produce executable machine code for a given device (e.g., GPUs) by translating the fragment implementation into a computational graph, which enables code optimizations.

The communication between fragments is also handled by the execution backends. For example, a DNN engine

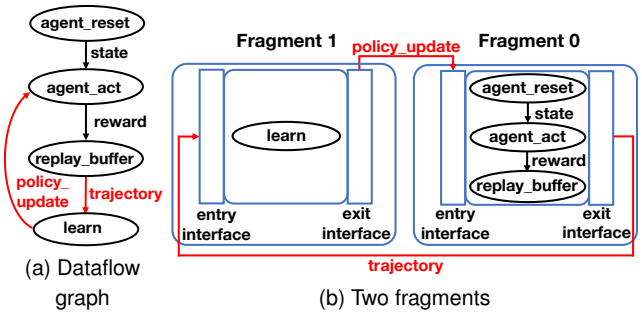


Fig. 5: Example of FDG generation for MAPPO

	MSRL	RLlib	WarpDrive
PPO	207	347 (+68%)	400 (+93%)
A3C	267	428 (+60%)	*n/a*

Tab. 4: Lines of code for the RL algorithm implementations

uses communication operators as part of its computational graph, automatically selecting suitable implementations (e.g., NCCL [39] for GPU collective communication).

Each worker has two components:

The **Fragment Optimizer** optimizes fragments that have been received for a given execution backend. To avoid the overhead of executing multiple instances of a replicated fragment, the optimizer attempts to *fuse* instances represented as computational graphs: it exploits the support of DNN engines to process data in a SIMD fashion by batching tensors from multiple fragment instances.

The Optimizer performs this transformation on the fragment's AST before submitting it to the DNN engine. It locates the AST nodes of tensors and merges their data. It then computes the new tensor shape to create a single tensor that can be processed by other data-parallel operators.

The **Executor backends** execute the fragments on a given target device: a DNN engine (MindSpore) executes computational graph on GPUs or CPUs; a CUDA job scheduler runs CUDA kernels on GPUs; a Python interpreter executes Python fragments on CPU cores; and a container scheduler can run arbitrary compute containers on CPU cores.

6 Evaluation

Our experimental evaluation answers the following questions: (i) what is the training performance that MSRL with FDGs achieves compared to existing RL systems with fixed execution strategies (§6.2)?; (ii) how does MSRL benefit from choosing different distribution policies (§6.3)?; and (iii) how well does MSRL scale in terms of the number of agents and the amount of training data (§6.4)?

6.1 Experimental set-up

Implementation. We implement MSRL in 11,700 lines of Python and C++ code. It uses CUDA 11.03, cuDNN 8.2.1, OpenMPI 4.0.3, and the MindSpore DNN framework 1.8.0 [19] as a GPU-based execution backend. The source code is available at https://github.com/mindspore-lab/mindrl.

Cluster	CPU cores *#nodes × #per node*	GPUs *#nodes × #per node*	Interconnects *intra-, inter-node*
Azure VMs NC24s_v2	Intel Xeon E5-2690 16×24, 448 GB	NVIDIA P100 16×4	PCIe 10 GbE
Local cluster	Intel Xeon 8160 4×96, 250 GB	NVIDIA V100 4×8	NVLink 100 Gbps IB

Tab. 5: Testbed configuration

MSRL uses the following distribution policies from Appendix A: DP-SingleLearnerCoarse; DP-SingleLearnerFine; DP-MultiLearner; DP-GPUOnly; and DP-Environments.

Baseline comparisons. For comparison, we use RLlib [34] of Ray V2.0, as a representative distributed RL system, and WarpDrive V1.6 [23], as a single-GPU system that accelerates the full RL training loop. Note that the implementations of RLlib Flow [26] and PodRacer [16] are unavailable.

RL algorithms. We focus on three popular algorithms: (1) *proximal policy optimization* (PPO) [47]; its multi-agent version, (2) *multi-agent PPO* (MAPPO) [57]; and (3) *asynchronous advantage actor-critic* (A3C) [31].

Tab. 4 compares the lines of code for the algorithm implementations. The RLLib and WarpDrive implementations require 68% and 93% more lines than MSRL, respectively, due to the hardcoded execution and distribution logic. This shows a benefit of MSRL's approach, which allows users to focus on the algorithm logic in their implementations.

For environments, we use two games (CartPole, HalfCheater) from the MuJoCo simulation engine [52], and two strategies (Spread, Tag) from the *multi-agent particle environment* (MPE) [28]. The policies use a 7-layer DNN.

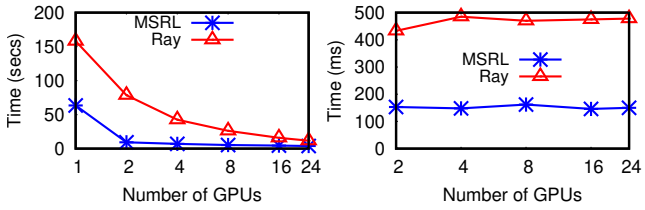
Testbeds. We conduct experiments on a *cloud* and a *local* cluster. The hardware details are given in Tab. 5: the cloud cluster has 16 VMs (with 64 GPUs); the local cluster has 4 nodes (with 32 GPUs). All nodes run Ubuntu Linux 20.04.

Metrics. For PPO, we measure (i) the training time to reach a given reward and (ii) the time per episode. For MAPPO, as the problem size increases with agents, we report (i) training time against the problem size and (ii) training throughput.

6.2 Performance with FDGs against baselines

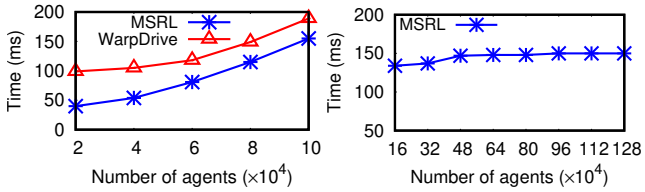
We investigate the performance impact that MSRL's FDG abstraction incurs compared to RL systems that only support hardcoded parallelization and distribution approaches.

Distributed training. We compare MSRL with DP-SingleLearnerCoarse to Ray [34] using PPO and A3C on the local cluster. For Ray, both algorithms are implemented using RLlib-Flow [26] and tuned based on RLlib's public PyTorch implementation [44]. DP-SingleLearnerCoarse is equivalent to the distribution approach implemented by RLlib-Flow's PPO and A3C implementations.



(a) Episode time vs. GPUs (PPO) (b) Episode time vs. GPUs (A3C)

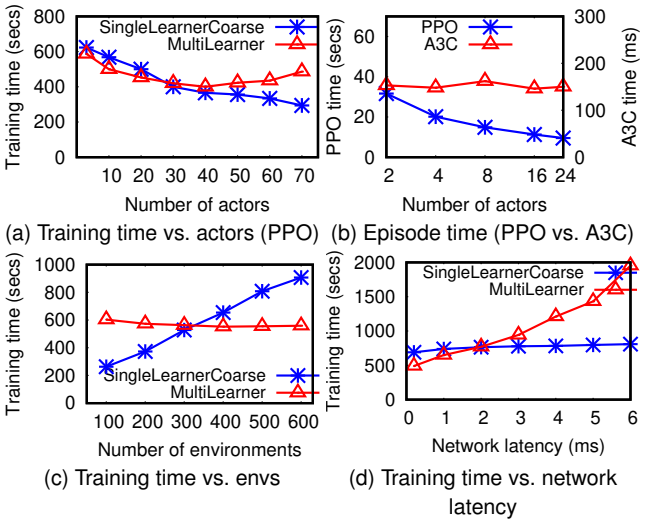
Fig. 6: Performance comparison with Ray



(a) Episode time vs. agents (1 GPU)

(b) Episode time vs. agents (n GPUs)

Fig. 7: Performance comparison with WarpDrive (PPO)



(a) Training time vs. actors (PPO) (b) Episode time (PPO vs. A3C)

(c) Training time vs. envs (d) Training time vs. network latency

Fig. 8: Impact of parameters on distribution policies

For PPO, we distribute 320 environments evenly among the actors, i.e., each actor interacts with $320/\#actors$ environments. A single learner trains the DNN. For A3C, one learner performs gradient optimization with gradients collected asynchronously from actors. Each actor interacts with one environment and computes gradients locally. We measure the time per episode, which is dominated by actor and environment execution. Since the DNN training/inference time is negligible, the fact that MSRL and Ray use different DNN frameworks (MindSpore vs. PyTorch) has low impact.

Fig. 6a shows the time per episode for PPO. MSRL's time with 1 GPU is 2.5× faster than Ray's, because Ray's CPU actor interacts with all environments sequentially. As the number of GPUs increases, both systems reduce episode time, because each actor interacts with fewer environments. With 24 GPUs, it takes 3.9 s for MSRL to execute an episode compared to 11.4 s for Ray (3× speed-up). When actors interact with multiple environments, MSRL combines DNN inference into one operation through FDG fusion, increasing GPU parallelism. It also uses fragments to execute environment steps in parallel by launching multiple processes.

Fig. 6b shows the time per episode for A3C. Both systems exhibit constant time with more GPUs, because the workload of each GPU executing an actor remains unchanged. MSRL executes actors 2.2× faster than Ray: since its distribution policy exploits customized asynchronous send/receive operations from the DL engine, it can avoid further data copies between GPUs and CPUs. In contrast, Ray must copy data to the CPU to communicate asynchronously.

In addition, MSRL generates the FDG that can be translated into a computational graph by the DL engine, enabling code optimizations and efficient execution. By leveraging code templates, MSRL generates optimized fragment code by directly manipulating the FDG AST.

GPU only training. Next, we use MSRL to deploy PPO with distribution policy DP-GPUOnly, which fuses the training loop into a single GPU fragment and replicates it for distributed execution. We use the *simple tag* MPE environment [28], which simulates a predator-prey game in which chaser agents are rewarded for catching runner agents. We train different numbers of agents, thus increasing the number of environments, on the local cluster and measure the training time per episode. We compare against WarpDrive [23], which performs single-GPU end-to-end RL training.

Fig. 7a shows the training time on 1 GPU. Compared to WarpDrive, MSRL is 1.2–2.5× faster when ranging from 20,000 to 100,000 agents. MSRL's DL engine (MindSpore) compiles fragments to computational graphs, exploiting more parallelization and optimization opportunities than WarpDrive's hand-crafted CUDA implementation.

While WarpDrive cannot scale to more than 1 GPU, Fig. 7b shows MSRL's performance when using up to 16 GPUs (each GPU trains 80,000 agents). Initially, training time increases from 138 ms to 150 ms due to the increased computation on a single worker (i.e., up to 640,000 agents). After that, training time is stable, and it is limited by communication bandwidth (InfiniBand, NVLink).

Conclusions: MSRL's FDG abstraction provides distribution policies for PPO and A3C that are tailored to their bottlenecks, e.g., enabling parallel environment execution and aggressively parallelizing GPU execution. Ray is limited by the distribution approach of its RLlib library; WarpDrive's manual CUDA implementation prevents it from exploiting more sophisticated compiler optimizations.

6.3 Trade-offs between distribution policies

Next, we explore the trade-offs between different distribution policies when changing RL algorithms and resources.

Actors. We measure PPO's training time with two distribution

policies, DP-SingleLearnerCoarse and DP-MultiLearner. We use a reward of 3,000 with 200 environments.

Fig. 8a shows the training time with 2 to 70 actors. DP-MultiLearner outperforms DP-SingleLearnerCoarse with fewer than 30 actors, but DP-SingleLearnerCoarse scales better after that, converging faster with more actors. Since DP-SingleLearnerCoarse only has 1 learner, its training batch size is fixed. Adding more actors therefore only distributes environment execution. In contrast, DP-MultiLearner fuses actors and learners into single fragments. With more actors, it also adds learners, reducing the batch size for each learner. This adds randomness to the training, affecting convergence [17].

Next, we compare two algorithms, PPO and A3C, under the same distribution policy DP-SingleLearnerCoarse.

Fig. 8b shows the time per episode for up to 24 actors. For PPO, the time decreases with the actor count; in contrast, A3C's time stays constant. Adding actors in PPO increases the parallelism of environment execution, thus reducing the workload per actor; for A3C, each actor only interacts with one environment, which makes its workload independent of the actor count. To reduce the episode time for A3C, a new distribution policy could be written that distributes the actor among multiple devices, combining data- or task-parallelism.

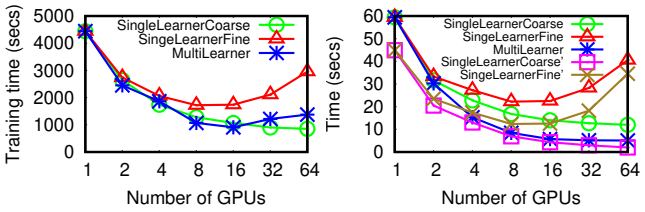
Environments. We explore how changing the number of environments affects the choice of distribution policy. When an agent interacts with more environments in parallel per episode, it trains with more data, improving training performance.

Fig. 8c shows the training time with 50 actors with 100–600 environments under DP-SingleLearnerCoarse and DP-MultiLearner. DP-MultiLearner scales better than DP-SingleLearnerCoarse with more than 320 environments: DP-SingleLearnerCoarse's training time increases with more environments, because its actors send trajectories to the learner, adding communication overhead; DP-MultiLearner only communicates gradients, having a fixed overhead.

Network latency. We examine the behavior of DP-SingleLearnerCoarse and DP-MultiLearner with PPO under different network latencies. We change network latency in our cloud cluster using the Linux traffic control (`tc`) tool from 0.2 ms to 6 ms. We use 400 environments and 50 actors.

As Fig. 8d shows, DP-MultiLearner is more sensitive to network latency than DP-SingleLearnerCoarse, and its training time increases with higher latency: since DP-MultiLearner uses Mindspore's data parallel model [19] to broadcast, aggregate and update gradients, it transmits many small tensors. This makes it a more suitable choice for cluster with low latency (< 2 ms); DP-SingleLearnerCoarse transmits the trajectory and DNN model weights as large tensors, performing data transmissions less frequently.

Cluster size. Finally, we study the performance of PPO under 3 distribution policies when increasing the GPU count: DP-SingleLearnerCoarse and DP-SingleLearnerFine use a single learner but apply different synchronization granularities; DP-MultiLearner scales to multiple learners using data-parallelism.



(a) Training time vs. GPUs (b) Episode time vs. GPUs

Fig. 9: Impact of GPU count on distribution polices

We use a constant 320 Mujoco HalfCheetah [4] environments.

Fig. 9a shows the training time in the cloud cluster to reach a reward of 4,000 with up to 64 GPUs; Fig. 9b reports the time per episode. With 64 GPUs, DP-SingleLearnerCoarse achieves the best speed-up in training time (5.3×). It maintains local copies of the DNN model at the actor and learner, and only actors send the batched states and rewards to the learner at the end of each episode (i.e., after 1,000 steps). This reduces the overhead with more GPUs compared to DP-SingleLearnerFine, whose actor fragments must communicate with the learner at each step.

DP-MultiLearner exhibits a different behavior: with 16 GPUs, it achieves better performance than either DP-SingleLearnerCoarse and DP-SingleLearnerFine, because it distributes policy training: it trains smaller trajectory batches on each GPU and aggregates the gradients from all GPUs. Instead, DP-SingleLearnerFine and DP-SingleLearnerCoarse gather all batches and train them using 1 learner.

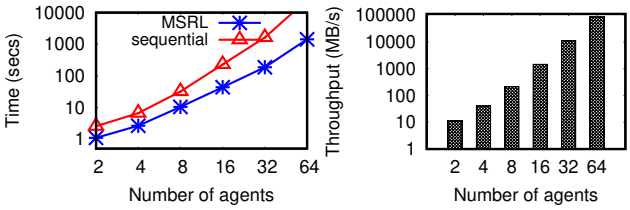
With more than 16 GPUs, DP-MultiLearner performs worse than DP-SingleLearnerCoarse: batches become smaller, making the gradient aggregation less efficient compared to training a large batch. Although DP-MultiLearner trains each episode faster than DP-SingleLearnerCoarse (see Fig. 9b), it requires more episodes to reach a similar reward value.

Note that DP-SingleLearnerCoarse and DP-SingleLearnerFine use the original PPO implementation with 1 learner [47], which limits scalability due to the centralized policy training (❸ in Fig. 1). To ignore this bottleneck in the algorithm, Fig. 9b also reports only the policy training time (labelled DP-SingleLearnerCoarse' and DP-SingeLearnerFine'). Now, MSRL scales better: when moving from 32 to 64 GPUs, performance increases by 25%.

Conclusions: As hyper-parameters, network properties or GPU counts change, the differences between distribution policies in terms of synchronization granularity and frequency of impact performance. MSRL's ability to allow users to switch between distribution policies at deployment time means that they can achieve the best performance in different scenarios without changing the algorithm implementation.

6.4 Scalability

Finally, we investigate how MSRL's design scales with the number of deployed agents for a MARL algorithm and of environments, thus increasing training data. We want to validate



(a) Training time per episode (b) Training throughput

Fig. 10: Scalability with agent count (MAPPO)

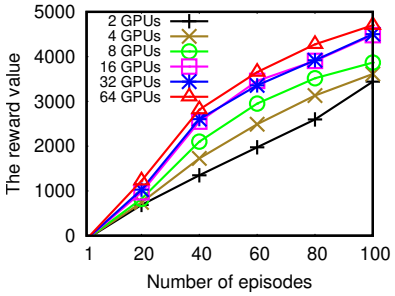


Fig. 11: Statistical efficiency with environment count (PPO)

if MSRL's approach introduces scalability bottlenecks.

Agents. We use MAPPO with the MPE *simple spread* environment [28], in which n agents learn to cover n landmarks while avoiding collisions. Agents must also process global observations on how far the closest agent is to each landmark. This results in $O(n^3)$ observations with n agents, quickly growing in computational cost and GPU memory usage [28]. We deploy on the cloud cluster using DP-Environments: each GPU trains 1 agent, and 1 worker executes all environments.

Fig. 10a shows the training time per episode for up to 64 GPUs against a sequential baseline (1 GPU). Due to its cubic complexity, the time increases both for the baseline and MSRL. With distributed training, MSRL's time grows more slowly than the baseline: with 32 agents, MSRL improves performance by 58×; with 64 agents, the baseline exhausts GPU memory, while MSRL trains one episode in 23.8 mins.

Fig. 10b compares the throughput with different agent numbers. Throughput is measured as the amount of data trained per second (in MB/s). Adding more agents (i.e., GPUs) significantly improves throughput, and the margin becomes larger with more GPUs: the throughput with 64 agents is over 7,600× higher than with 2 agents, as multiple GPUs train agents in parallel, processing more observations per GPU.

Environments. We observe the effect of more environments on statistical efficiency, i.e., the episodes needed to reach a given reward. We use 10 environments per CPU, adding more workers in the cloud cluster using DP-SingleLearnerCoarse.

Fig. 11 shows the reward along with the number of episodes for different environment counts. More environments lead to a higher reward: as more CPUs execute environments, the larger use of trajectories per episode yields a higher reward.

Conclusions: FDGs do not deteriorate scalability. MSRL scales to a large number of data-intensive agents, handling the increase in communication between fragments without bottlenecks. A larger amount of data generated by more environments also improves the statistical efficiency of training.

7 Related Work

DNN compilation. XLA [56] is a domain-specific compiler that accelerates the linear algebra of DNN models. JAX [12] uses just-in-time (JIT) compilation to transform vectorized Python programs to GPU or TPU code. TVM [6] is an automated end-to-end optimizing compiler for DNN training. These approaches focus on DNN training and inference workloads with regular computation/communication patterns. In contrast, RL algorithms exhibit more complex control and data flow in their training loops.

DNN auto-parallelization. Alpa [60] and Unity [53] automatically parallelize and distribute DNN training using data/operator/pipeline parallelism. They search for effective distributed execution plans using dynamic or integer linear programming. In future work, we want to explore the use of optimization techniques to generate an optimal distribution policy for a given RL algorithm. Since an FDG has more heterogeneity than DNN dataflows, the search space is substantially larger and based on more complex cost models.

Dataflow and actor systems. Spark [58] and Naiad [36] express programs as dataflow graphs, sharding data across workers. They provide high-level APIs to express computation as a single homogeneous dataflow. In contrast, FDGs allow different dataflow models to be integrated into a single distributed computation, as governed by distribution policies.

Ray [34] offers a general actor-based platform for distributed computing. To support RL algorithms, it uses domain-specific libraries (RLlib/RayFlow [25, 26]) that hardcode distribution strategies, preventing it from switching strategies based on e.g., hardware properties. Instead, MSRL proposes FDG, a higher-level abstraction for parallelizing and distributing RL training loops, which decouples RL algorithms from their execution through explicit distribution policies.

8 Conclusions

While DNN systems have mature dataflow abstractions that improve execution performance, similar abstractions for RL systems have been under-explored. We described MSRL, a system that supports the flexible parallelization and distribution of RL algorithms using *fragmented dataflow graph*. Accounting for the heterogeneous nature of RL training loops, MSRL separates the algorithm from its execution by using *distribution policies* that allocate dataflow fragments to GPUs and CPUs. Our experiments showed how distribution policies generalize existing RL systems without overhead.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Yibo Zhu, for their helpful comments.

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 *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2016.

[2] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18:153:1–153:43, 2017.

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

[5] Michael Chang, Sidhant Kaushik, S. Matthew Weinberg, Tom Griffiths, and Sergey Levine. Decentralized reinforcement learning: Global decision-making via local economic transactions. In *Proceedings of the 37th International Conference on Machine Learning, ICML*, 2020.

[6] 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 *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.

[7] Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre M. Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2020.

[8] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: scalable and efficient deep-rl with accelerated central inference. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[9] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, 2018.

[10] Python Software Foundation. Process-based parallelism, 2022. [Online; accessed 10-December-2021].

[11] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021.

[12] Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *Systems for Machine Learning*, 2018.

[13] Sven Gronauer and Klaus Diepold. Multi-agent deep reinforcement learning: a survey. *Artificial Intelligence Review*, pages 1–49, 2021.

[14] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. [Online; accessed 25-June-2019].

[15] Abhishek Gupta, Justin Yu, Tony Z. Zhao, Vikash Kumar, Aaron Rovinsky, Kelvin Xu, Thomas Devlin, and Sergey Levine. Reset-free reinforcement learning via multi-task learning: Learning dexterous manipulation behaviors without human intervention. In *IEEE International Conference on Robotics and Automation, ICRA 2021*, 2021.

[16] Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *CoRR*, abs/2104.06272, 2021.

[17] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, 2017.

[18] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alexander Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Çaglar Gülçehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *CoRR*, abs/2006.00979, 2020.

[19] Huawei. Mindspore. https://www.mindspore.cn/en, 2020.

[20] Huawei. Mindspore all gather operator. https://www.mindspore.cn/docs/en/r1.7/api_python/ops/mindspore.ops.AllGather.html?highlight=allgather, 2022. [Online; accessed 18-May-2022].

[21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

[22] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems 12, [NIPS Conference]*, 1999.

[23] Tian Lan, Sunil Srinivasa, and Stephan Zheng. Warpdrive: Extremely fast end-to-end deep multi-agent reinforcement learning on a GPU. *CoRR*, abs/2108.13976, 2021.

[24] 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*, 2014.

[25] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, 2018.

[26] Eric Liang, Zhanghao Wu, Michael Luo, Sven Mika, and Ion Stoica. Distributed reinforcement learning is a dataflow problem. *CoRR*, abs/2011.12719, 2020.

[27] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR*, 2016.

[28] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, 2017.

[29] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter R. Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2020.

[30] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[31] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, 2016.

[32] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML*, 2016.

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533, 2015.

[34] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.

[35] Paul Muller, Shayegan Omidshafiei, Mark Rowland, Karl Tuyls, Julien Pérolat, Siqi Liu, Daniel Hennes, Luke Marris, Marc Lanctot, Edward Hughes, Zhe Wang, Guy Lever, Nicolas Heess, Thore Graepel, and Rémi Munos. A generalized training approach for multiagent learning. In *8th International Conference on Learning Representations, ICLR*, 2020.

[36] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13*, 2013.

[37] Ranjit Nair, Milind Tambe, Makoto Yokoo, David V. Pynadath, and Stacy Marsella. Taming decentralized pomdps: Towards efficient policy computation for multiagent settings. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 2003.

[38] NVIDIA. CUDA Toolkit: Develop, optimize and deploy gpu-accelerated apps, 2022. [Online; accessed 10-December-2021].

[39] NVIDIA. NCCL: Nvidia collective communications library, 2022. [Online; accessed 10-December-2021].

[40] NVIDIA. Nvlink and nvswitch. https://www.nvidia.com/en-au/data-center/nvlink, 2023. [Online; accessed 10-Sep-2022].

[41] OpenAI. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt, 2022.

[42] 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 *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2019.

[43] Chao Qu, Shie Mannor, Huan Xu, Yuan Qi, Le Song, and Junwu Xiong. Value propagation for decentralized networked deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS*, 2019.

[44] Ray. Ray ppo setting. https://github.com/ray-project/ray/blob/master/rllib/tuned_examples/ppo/halfcheetah-ppo.yaml, 2020. [Online; accessed 10-Sep-2022].

[45] Michael Schaarschmidt, Sven Mika, Kai Fricke, and Eiko Yoneki. RLgraph: Modular Computation Graphs for Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems, MLSys*, 2019.

[46] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[47] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[48] Tom Shanley. *Infiniband*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[49] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

[50] Peng Sun, Jiechao Xiong, Lei Han, Xinghai Sun, Shuxing Li, Jiawei Xu, Meng Fang, and Zhengyou Zhang. Tleague: A framework for competitive self-play based distributed multi-agent reinforcement learning. *CoRR*, abs/2011.12895, 2020.

[51] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[52] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[53] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2022.

[54] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019.

[55] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, 1992.

[56] XLA and TensorFlow teams. XLA: Optimizing compiler for machine learning, 2022. [Online; accessed 10-December-2021].

[57] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and YI WU. The surprising effectiveness of ppo in cooperative multi-agent games. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2022.

[58] 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, 2016.

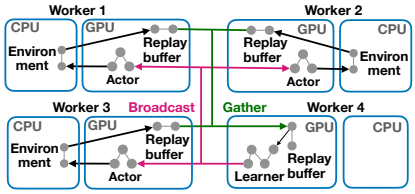
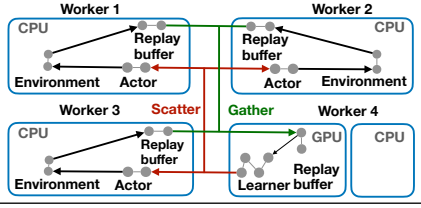
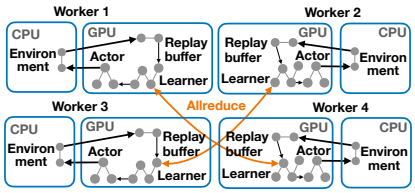
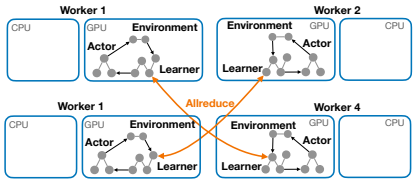
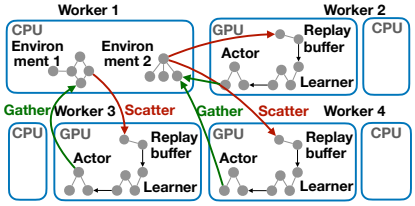
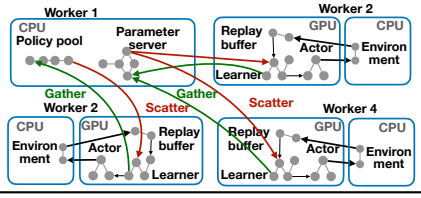
[59] Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. Fully decentralized multi-agent reinforcement learning with networked agents. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, 2018.

[60] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, and Joseph E Gonzalez. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

[61] Ming Zhou, Ziyu Wan, Hanjing Wang, Muning Wen, Runzhe Wu, Ying Wen, Yaodong Yang, Weinan Zhang, and Jun Wang. Malib: A parallel framework for population-based multi-agent reinforcement learning. *CoRR*, abs/2106.07551, 2021.

[62] Matthieu Zimmer, Claire Glanois, Umer Siddique, and Paul Weng. Learning fair policies in decentralized cooperative multi-agent reinforcement learning. In *Proceedings of the 38th International Conference on Machine Learning, ICML*, 2021.

A Supported Distribution Policies

Distribution policy	Deployment	Description
[DP-SingleLearnerCoarse] replicate: (*actor*, *env*) split: (*learner*) e.g., Acme [18], Sebulba [16]		DP-SingleLearnerCoarse replicates the actor and environment fragments: W1–W3 co-locate 1 GPU fragment with an actor for DNN policy inference and 1 CPU fragment for the environment execution. A single GPU fragment with a learner performs policy training (W4), gathering batched training data, training the policy and broadcasting updates.
[DP-SingeLearnerFine] replicate: fused *actor/env* split: *learner* e.g., SEED RL [8]		DP-SingleLearnerFine fuses the actor and environment into 1 fragment (W1–W3) but handles policy inference at the learner (W4), i.e., actors do not contain DNNs. W4 executes policy inference and training in 1 GPU fragment; W1–3 only have CPU fragments. W4 scatters actions to W1–W3 and gathers data for policy training.
[DP-MultiLearner] replicate: fused *actor/learner*, *env*		DP-MultiLearner performs data-parallel training with multiple learners, supporting fully decentralised MARL training [5,43,59,62]. DP-MultiLearner co-locates 2 fragments: a GPU fragment that fuses the actor and learner, accelerating policy inference, training and replay buffer management, and a CPU fragment for environment execution.
[DP-GPUOnly] replicate: fused *actor/learner/env*		DP-GPUOnly fuses the training loop into 1 GPU fragment. To enable communication among GPU fragments, DP-GPUOnly uses `Allreduce` operators compiled into the computational graph with NCCL2 [39]. DP-GPUOnly is a distributed implementation of the single-node systems (e.g., WarpDrive [23]).
[DP-Environments] replicate: fused *actor/learner* split: *env* e.g., MALib [61]		DP-Environments has a dedicated worker for environment execution. W1 has CPU fragments to execute environment instances on multiple CPU cores; W2–W4 fuse the actor and learner to accelerate policy inference and training. W1 gathers the inferred actions and scatters the states and rewards.
[DP-Central] replicate: fused *actor/learner*, *env* split: *param server/policy pool*		DP-Central supports a central *policy pool* [61] or *parameter server* [24] on a separate worker (W1). W2–W4 co-locate GPU fragments for policy inference and training and CPU fragments for environment execution.

Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent

Qizhen Weng[†*] Lingyun Yang[†*] Yinghao Yu[§†] Wei Wang[†]
Xiaochuan Tang[§] Guodong Yang[§] Liping Zhang[§]

[†]Hong Kong University of Science and Technology [§]Alibaba Group
{qwengaa, lyangbk, weiwa}@cse.ust.hk, {yinghao.yyh, xiaochuan.txc, liping.z}@alibaba-inc.com, luren.ygd@taobao.com

Abstract

Large tech companies are piling up a massive number of GPUs in their server fleets to run diverse machine learning (ML) workloads. However, these expensive devices often suffer from significant underutilization. To tackle this issue, GPU sharing techniques have been developed to enable multiple ML tasks to run on a single GPU. Nevertheless, our analysis of Alibaba production traces reveals that allocating partial GPUs can result in severe *GPU fragmentation* in large clusters, leaving hundreds of GPUs unable to be allocated. Existing resource packing algorithms fall short in addressing this problem, as GPU sharing mandates a new scheduling formulation beyond the classic bin packing.

In this paper, we propose a novel measure of fragmentation to statistically quantify the extent of GPU fragmentation caused by different sources. Building upon this measure, we propose to schedule GPU-sharing workloads towards the direction of *the steepest descent of fragmentation*, an approach we call *Fragmentation Gradient Descent* (FGD). Intuitively, FGD packs tasks to minimize the growth of GPU fragmentation, thereby achieving the maximum GPU allocation rate. We have implemented FGD as a new scheduler in Kubernetes and evaluated its performance using production traces on an emulated cluster comprising more than 6,200 GPUs. Compared to the existing packing-based schedulers, FGD reduces unallocated GPUs by up to 49%, resulting in the utilization of additional 290 GPUs.

1 Introduction

Graphics Processing Units (GPUs) are widely deployed in production clusters to accelerate machine learning (ML) tasks for a plethora of AI applications [14, 16, 17, 21, 31, 39]. Compared to CPUs and other resources, GPUs are considerably more expensive but often underutilized in production clusters, with the reported utilization rates ranging from 25% to below 50% [16, 19, 22, 31].

The primary reason for low GPU utilization is that a large number of ML tasks, mostly inference, cannot fully utilize the capacities of modern GPUs, which have seen exponential performance improvements in recent years. This trend is expected to continue in the foreseeable future [36]. To address this issue, GPU sharing techniques have been developed to enable multiple ML tasks to safely run on a single GPU with guaranteed isolation, where each task is allocated *partial resources* by means of virtualization [9, 13, 27, 33, 35] or the Multi-Instance-GPU (MIG) feature supported in NVIDIA's Ampere architecture [3].

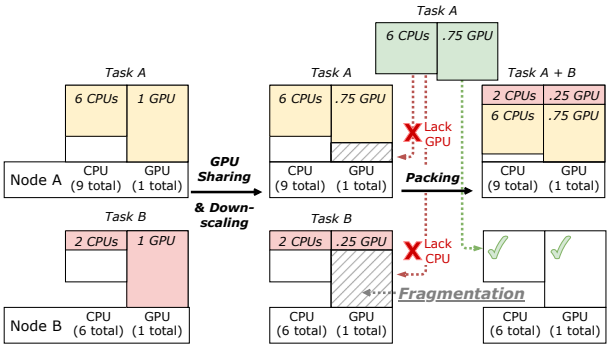


Figure 1: The allocation of partial GPUs results in fragmentation, which can be addressed with packing.

However, simply enabling GPU sharing does not necessarily lead to high utilization. In many cases, allocating partial GPUs results in *fragmentation*, preventing the remaining GPU resources from being allocated. Figure 1 illustrates this problem in a toy example. Consider a cluster of two nodes A and B with ⟨9 CPUs, 1 GPU⟩ and ⟨6 CPUs, 1 GPU⟩, respectively. There are two tasks A and B running on the two nodes, each demanding ⟨6 CPUs, 0.75 GPU⟩ and ⟨2 CPUs, 0.25 GPU⟩, respectively. Without GPU sharing, both tasks are allocated an entire GPU even though they cannot fully utilize it (Figure 1, left). This problem can be addressed by allocating partial GPUs, using the GPU sharing technique (Figure 1, middle). Now supposing another instance of task A arrives, it cannot run on either node even though the cluster has sufficient aggregate GPU resources ($0.25 + 0.75 = 1$ GPU).

GPU fragmentation has been widely observed in our production clusters that support GPU sharing. Figure 2 shows a 7-day trace collected from a 1280-GPU cluster. On average, the aggregate GPU allocations account for 77.6% of the total

*Equal contributions.

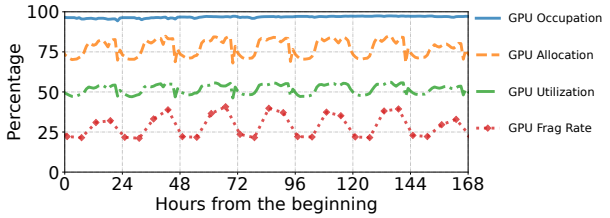


Figure 2: A 7-day trace from a large GPU-sharing cluster. GPU occupation measures the number of GPU devices that are not fully idle; GPU allocation measures the total amount of allocated GPU resources; GPU utilization refers to the proportion of actual GPU resources used by tasks; GPU frag rate is the percentage of unallocated GPU resources that become unusable due to fragmentation (defined in §3.2).

capacity (orange dashed line). These allocations, many being partial GPUs, are distributed across almost all GPU devices (blue solid line), turning 21–42% of the unallocated GPU resources into fragments (red dotted line) that cannot be utilized by the current workload. In general, higher allocations result in more severe fragmentation. In our operational experience, complaints about scheduling failures usually surge in rush hours, although the cluster still has sufficient aggregate idle GPUs, indicating severe fragmentation.

An effective approach to addressing fragmentation is to perform *packing*. Returning to the previous example, the scheduler can instead pack the two original tasks to node A, thereby leaving the entire node B to the new instance of task A (Figure 1, right). A large body of work formulates workload scheduling as a multi-dimensional bin packing problem, in which tasks and nodes are respectively modeled as balls and bins with sizes in multiple resource dimensions, and the goal is to pack balls to the fewest number of bins. Many heuristics have been proposed to schedule cluster workloads, such as best-fit [11, 21, 24, 30], vector alignment scoring [8, 23, 24], and "GPU Packing" [31, 33].

However, our experiments show that none of these heuristics work well in scheduling GPU-sharing workloads (§6). The fundamental reason, we believe, is that the problem is *intrinsically different* from the classic bin packing when a node has multiple GPUs. Consider two natural bin packing formulations. The first is to model a server's multiple GPUs as one large device with the aggregate capacity. This formulation pools together all unallocated GPUs, and the fragments on individual GPUs become irrelevant, which is not the case in reality. Alternatively, one can treat a server's multiple GPUs as different "resource dimensions". Yet, unlike other resources such as CPU and memory, these GPUs are not independent but *interchangeable* for a task that can run on any of them, mandating a new formulation beyond classic bin packing.

In this paper, we develop a novel fragmentation-aware scheduling approach for GPU-sharing workloads. Central to our approach is a new analytical framework that quantifies statistically the degree of GPU fragmentation in a cluster. Given a task, we identify the GPUs on each node that cannot be used to run the task (e.g., lacking sufficient GPU or other resources). These GPUs are *fragmented* from the view of that task as none of their remaining resources can be utilized. Now, consider the *target workload*, which consists of a set of tasks that are of interest (e.g., ML inference and training). We quantify the degree of GPU fragmentation as *the expected number of GPUs that cannot be allocated to a task which is randomly sampled from the target workload*. Intuitively, it measures the expected GPU resources that cannot be utilized by the target workload. We can further break down the fragmentation analysis into different causes, such as the node having insufficient or stranded GPUs, or the mismatch between the workload and the node spec. This analysis provides more insights for the operator to reason about the cluster state (§3).

Based on the GPU fragmentation analysis, we propose a simple, yet effective heuristic to schedule workloads towards the direction of *the steepest descent of fragmentation*, which we call Fragmentation Gradient Descent or FGD. For each GPU task submitted, FGD chooses a node and the available GPU(s) on it to run the task so that the growth of GPU fragmentation caused by this decision is minimized (§4). By doing so, FGD can minimize GPU fragmentation, saving a large amount of expensive resources for more workloads.

We have implemented FGD as a new scheduler in Kubernetes [1] (§5) and evaluated its performance with production and synthesized workload traces on an emulated cluster consisting of more than 1,200 nodes and 6,200 GPUs (§6). FGD consistently outperforms existing packing-based scheduling algorithms in various settings: it reduces the unallocated GPUs by up to 49%, allowing 290 GPUs to be utilized in a large production cluster. Our implementation, including the scheduler and the emulator[1], as well as the trace data[2] used in the evaluation, are available as open-source software.

2 Background and Motivation

In this section, we briefly introduce the GPU sharing technique and illustrate the resulted fragmentation problem through production trace analysis. We discuss the unique scheduling challenge brought by GPU sharing that invalidates the classic bin packing formulation.

2.1 GPU Sharing

Underutilized GPUs. GPU underutilization has been widely observed in production clusters that run diverse ML workloads. Many tech companies report the low GPU utilization averaging between 25% to below 50% [16, 19, 22, 31], which has become a thorny pain point in reducing the total cost of ownership of large GPU clusters.

[1] https://github.com/hkust-adsl/kubernetes-scheduler-simulator
[2] https://github.com/alibaba/clusterdata

There are multiple factors that contribute to the low GPU utilization. Most importantly, thanks to the exponential improvement of GPU performance in recent years, many ML tasks, especially inference, cannot saturate the compute capacity of a modern GPU. Taking the latest A100 GPU as an example, the peak inference speed of a ResNet50 [15] model reaches over 36k images per second [36], far exceeding the usual throughput requirement of an object detection application. In fact, even for training tasks, increasing evidences show that many of them cannot fully utilize a GPU [5, 6, 28, 31, 33, 34, 37].

GPU Sharing. GPU sharing techniques have recently been developed to enable multiple tasks to run on a single GPU with guaranteed performance isolation, where each task is allocated a partial GPU. In production systems, GPU sharing can be implemented at three levels.

1) Framework-level: This approach adds new dynamic scaling mechanisms and sharing primitives to the existing ML frameworks (e.g., TensorFlow, PyTorch, JAX) to allocate each task the exact amount of required GPU memory and compute units [6, 33, 34, 37]. The benefit of this approach is that it can achieve fine-grained sharing between tasks by leveraging their semantics information (e.g., training accuracy and loss), which is available to the framework. On the other hand, it requires users to switch to the modified framework to enable GPU sharing – not all users are willing to do so.

2) Device runtime-level: This approach uses the API remoting technique to implement GPU sharing and virtualization [9, 13, 27, 28, 32]. It deploys a GPU manager on each host. The manager intercepts compute- and memory-related runtime APIs (e.g., `cuLaunchKernel` and `cuMemAlloc` in CUDA Library) to keep track of the compute and memory resources requested by each task. A task's memory allocation request is accepted only when the its allocation is within the specified limit. The manager also controls kernel scheduling to enforce the specified allocation of compute capacity by time-multiplexing the device's compute units between tasks (e.g., a task with 0.1 GPUs is guaranteed to receive at least 10% of GPU time). This approach requires no framework changes or user cooperation.

3) Hardware-assisted: Starting with Ampere architecture, NVIDIA GPUs support the Multi-Instance GPU (MIG) feature. MIG can partition a GPU into as many as seven separate instances [3]. Compared with the software approaches, MIG provides the strongest isolation guarantee as each GPU instance has dedicated resources for compute, memory, and memory bandwidth. On the downside, MIG only supports resource sharing at a coarse granularity and is available exclusively to A100, A30, and H100 GPUs at the moment.

Production Deployment in Our Clusters. At Alibaba, we have developed our own GPU sharing solution based on CUDA Runtime API interception and deployed it in production clusters that run a mixture of training and inference tasks. A task can have one or multiple instances, each running in a container and requesting multiple resources such as CPUs, memory, and GPUs. We observe the GPU requests to be either a partial GPU or full GPU(s), but rarely the combination of both (e.g., 1.5 or 2.3 GPUs). In our implementation, the minimum GPU allocation unit is 0.01 GPUs, in which a task instance is allocated 1% of the GPU memory and *at least* 1% of the GPU time.[3] Tasks and their instances are orchestrated using a customized Kubernetes system [1] with many new features tailored to the production needs. In the following discussions, unless otherwise specified, *we do not differentiate between tasks and instances* as their meanings are usually clear from the context.

2.2 The Prevalence of GPU Fragmentation

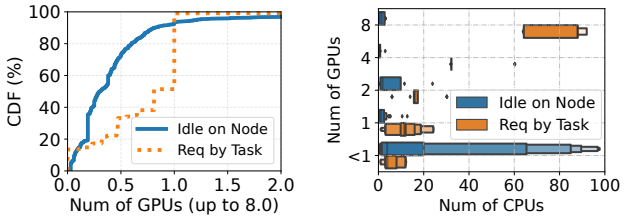
Operational Experience. GPU sharing greatly reduces the number of allocated GPUs for a workload, allowing more tasks to run in a cluster than before. However, as we consolidate more workloads, an increasing number of users complained about the long task wait time or even scheduling failures due to the timeout of pending tasks, although they still have sufficient GPU quotas to spare. In many clusters, the GPU allocation rate can reach 85–90% maximum, leaving hundreds of GPUs unable to utilize. In some extreme cases, pending tasks start to build up when the GPU allocation rate reaches above 80%. All these indicate heavy fragmentation.

Trace Analysis. We perform trace analysis to confirm the fragmentation problem, which occurs when a GPU has *insufficient resources* or becomes *stranded* as the host runs out of the other resources, such as CPU and memory. We choose a highly crowded ML cluster H consisting of 1.2k nodes with over 6.2k GPUs and 107k CPU cores. After scheduling over 7.6k tasks, the GPU (CPU) allocation ratio reaches 92% (75%), which is among the highest in all clusters. At this point, cluster H is fully packed and cannot accommodate new tasks despite having a total of 500 GPUs unallocated, causing a significant resource waste. Figure 3a depicts the distributions of unallocated GPUs on each node and the requested GPUs of each pending task. Around 92% of nodes have < 1 GPU left unallocated (blue solid line), whilst 49% of the pending tasks request ≥ 1 GPU (orange dashed line). Figure 3b gives the boxen plot of the nodes' unallocated CPUs and the tasks' requested CPUs, grouped by their GPU resources. Among the nodes with ≥ 1 unallocated GPU, over 75% have ≤ 10 CPUs left (blue boxes), which are *insufficient* to allocate to most tasks (orange boxes), leaving the unallocated GPUs stranded.

2.3 Inapplicable Bin Packing Formulation

While fragmentation is not a novel problem in cluster management, GPU fragmentation is noteworthy because it (1)

[3] In our system, a task instance can opportunistically use more GPU time if the computing capacity of the said GPU is not exhausted by the other tasks.



(a) CDF of idle and requested GPUs. (b) Boxen of idle and requested CPUs.

Figure 3: Illustration of GPU fragmentation in a fully packed cluster H. (a) Most nodes have insufficient GPUs. (b) Nodes with abundant GPUs are usually in short of CPUs.

inherently differs from other resource fragmentation problem and (2) is aggravated by partial-GPU allocation (Figure 1).

GPU Fragmentation Cannot be Handled Similarly as Other Resources. Fragmentation is a common problem in resource allocation. Taking file allocation as an example, when the disk has no enough contiguous space to store a file, the fragmentation occurs [4], and the solution is to chunk the file into blocks for non-contiguous allocation. However, GPU allocation must be *contiguous*: Consider a task requesting one full GPU, it is not possible to allocate it two partial GPUs (e.g., 0.3 GPUs + 0.7 GPUs).

Partial-GPU Allocation Invalidates Bin Packing Formulation. Bin packing is known effective to address the fragmentation problem [23, 24]. In the standard formulation, tasks and nodes are respectively modeled as balls and bins of sizes in $\mathbb{R}^d$, where d is the number of concerned resources, such as CPU, memory, and GPU. The goal is to pack balls to as fewest bins as possible. However, unless a node has a single GPU, this formulation does not apply to GPU-sharing tasks, which we illustrate through two formulation attempts.

Attempt-1: Treating Multiple GPUs as a Unified Logical Device. This formulation pools together all the available GPU resources of a node into a large logical GPU, leading to a node resource vector such as $\langle 16\text{ CPUs}, 24\text{ GiB memory}, 1.3\text{ GPUs}\rangle$. However, this formulation is problematic as it ignores the *allocation boundary* of physical GPUs, making it unable to differentiate the fragmentation state on each individual device. For example, consider a 2-GPU node with the remaining capacity of 0.4 GPUs and 0.9 GPUs. For a task that requests one full GPU, although in total the node has 1.3 GPUs, neither of its two GPUs has sufficient capacity to run the task, to which both GPUs are fragmented.

Attempt-2: Treating Each GPU as an Independent Resource Dimension. An alternative formulation is to model each GPU on a node as an independent resource dimension, just like CPU and memory. This formulation is also problematic as GPUs are *interchangeable* as long as they have sufficient capacity to run a task. Returning to the previous example and assuming that the node has 16 CPU cores and 24 GiB memory available, its resource vector is $\langle 16\text{ CPUs}, 24\text{ GiB memory}, 0.4\text{ GPUs}, 0.9\text{ GPUs}\rangle$. For a task that requests 2 cores, 8 GiB memory, and 0.3 GPUs, it can run on the node with any of the two GPUs. The task hence has two interchangeable demand vectors $\langle 2\text{ CPUs}, 8\text{ GiB memory}, 0.3\text{ GPUs}, 0\text{ GPUs}\rangle$ or $\langle 2\text{ CPUs}, 8\text{ GiB memory}, 0\text{ GPUs}, 0.3\text{ GPUs}\rangle$. This invalidates the classic bin packing formulation as the balls are now "deformable" and can transform to various sizes.

To summarize, we stress that the inapplicability of bin packing formulation stems from the contiguous GPU allocation requirement and the partial-GPU allocation practice, in which each GPU has its own allocation boundary (Attempt 1) but is also interchangeable to one another (Attempt 2). Through extensive experiments in §6, we will show that none of the existing packing heuristics, including best-fit [11, 21, 24, 30], vector alignment scoring [8, 23, 24], and "GPU Packing" [31, 33], work well in scheduling GPU sharing workload.

3 The Fragmentation Measure

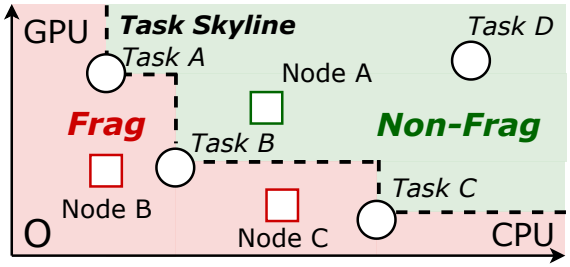
"You keep using that word. I do not think it means what you think it means." — Inigo Montoya, *The Princess Bride*

While the term fragmentation has been frequently mentioned in the existing cluster scheduling works [8, 11, 29, 30, 33, 34, 39, 40], its formal definition and quantitative measure remain unclear. In this section, we answer *what fragmentation is and how can it be measured*. We start with a conventional measure defined in absolute terms and discuss its ineffectiveness (§3.1). We next present a new measure that statistically quantifies the fragmentation degree in a cluster (§3.2). We show in case studies that the new measure can help operators better reason about the cluster state (§3.3).

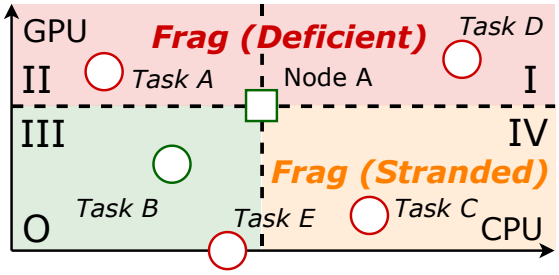
3.1 Fragmentation in Absolute Terms

Basic Assumption. In general, *whether a node has fragmented resources depends on the target workload*. For example, a node with 0.6 GPUs is considered fragmented by a task that requests one full GPU, but not by the one that demands 0.5 GPUs. In this paper, we assume that the target workload, which contains a set of tasks with popularity (e.g., number of instances) following a certain distribution, is known to the cluster. This is reasonable as production ML workloads consist of a large number of recurring tasks [31]. We also assume that each task can request either a partial GPU or full GPU(s), but not both, as mentioned in §2.1.

An Absolute Fragmentation Measure. Although not formally defined, it is commonly accepted that a node is fragmented *if its remaining resources cannot be allocated to run any tasks* in the target workload. In this definition, fragmentation is measured in absolute terms: regardless of task scheduling, the fragmented resources cannot be utilized anyway and are inevitably wasted.



(a) Absolute fragmentation defined by task skyline is defective.



(b) Statistical fragmentation in expectation of randomly sampled tasks.

Figure 4: Fragmentation definition in the example cluster with nodes of various unallocated resources (squares) and tasks of various requested resources (circles).

Figure 4a gives a pictorial illustration. For simplicity, we only consider CPU and GPU resources in a two-dimensional plane. A task is depicted as a circle (ball) with x- and y-coordinates being the requested CPUs and GPUs, respectively. Similarly, a node is depicted as a box (bin) with the two coordinates being the unallocated CPUs and GPUs. In case that a node has multiple GPUs, we map their unallocated capacity (a vector) to a scalar number as follows. Let f be the number of *fully-unallocated GPUs* and p the *maximum unallocated partial GPU*. We map the vector of unallocated GPUs to a scalar $u = f + p$.[4] For example, a 4-GPU node with an unallocated capacity of $\langle 1, 1, 0.5, 0.25 \rangle$ is considered having $u = 2.5$ unallocated GPUs. Under this mapping, a node has sufficient GPUs to run a task that requests g GPUs *if and only if* $u \geq g$, provided that the task requests either a partial GPU or full GPU(s), i.e., $g \in [0,1) \cup \mathbb{Z}^+$.

With nodes and tasks depicted in the resource plane, we see that a node has sufficient GPUs and CPUs to run a task if it is located above and on the right of the task (e.g., node A and task B). We call this region the *non-fragmentation region* of the task. Taking the union of the non-fragmentation regions of all tasks gives the non-fragmentation region of the target workload (the green area in Figure 4a). It encompasses all tasks, with the "innermost" ones (e.g., tasks A, B, and C) located on the region's border which we call the *skyline* [38]. The area below the skyline is the *fragmentation region*, within which a node cannot run any task due to insufficient resources and is thus fragmented (e.g., nodes B and C).

[4]The mapping is not unique. For example, alternatively one can map the capacity vector to $u = \max\{f, p\}$, which serves the same purpose.

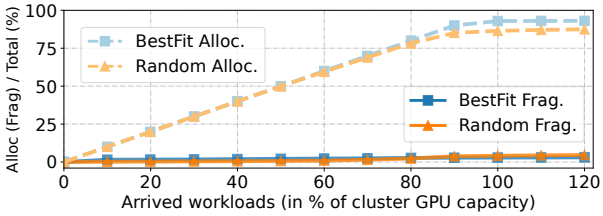


Figure 5: Trace-driven emulations with two scheduling policies (details in §6.1). The x-axis is the cumulative GPU demands of arrived tasks, divided by the total cluster capacity. Fragmentation in absolute terms (lower-right corner) stays at a low level ($< 5\%$) throughout the allocation of arrival workloads, failing to provide useful feedback to the scheduler.

The Inefficiency of the Absolute Measure. Under the absolute measure, resource fragmentation is identified in a rather biased manner. Whether a node is fragmented solely depends on if it has sufficient resources to run a task that is located on the skyline (i.e., the skyline task), whereas the other tasks are irrelevant. Yet, skyline tasks can rarely represent the entire workload. Compared with the other tasks, they request fewer CPUs and/or GPUs, and usually have a small population. In our clusters, only 0.06% of instances belong to the skyline tasks. On average, a skyline task requests $\langle$3.2 CPUs, 0.07 GPUs$\rangle$, as opposed to the average demand of $\langle$9.4 CPUs, 0.9 GPUs$\rangle$. As a result, even if a node has a small amount of resources that cannot be allocated to run the majority of the tasks in the target workload, it may still be considered non-fragmented as long as it can run a tiny skyline task.

For the reasons above, the absolute fragmentation measure cannot be used as a good metric to guide task scheduling. To see this, we run trace-driven emulations with two different scheduling policies (details in §6.1) and depict in Figure 5 the fragmented and allocated GPUs in percentage of the total capacity. The fragmentation measure stays at a low level ($< 5\%$) throughout the process regardless of the scheduling logic and its placement decisions, failing to provide useful feedback to the scheduler. Increased fragmentation is only observed when the cluster starts crowded, by which it is too late for the scheduler to take actions. In fact, even when the cluster becomes fully packed (i.e., the allocation curve flattens when the cumulative GPU demands reach over 100% of the cluster capacity), over 50% of unallocated GPUs are still deemed non-fragmented under the absolute measure.

3.2 A Statistical Fragmentation Measure

We believe a good fragmentation measure should not be defined against a small subset of tasks, but a joint calibration of the entire workload. We hence use a *statistical measure* to quantify the degree of fragmentation. Formally, let M be the target workload in which each task m has popularity p_m.

Without loss of generality, we assume normalized popularity where $\sum_{m \in M} p_m = 1$. Given a node n, $F_n(M)$ denotes the GPU fragmentation measured by workload M, and $F_n(m)$ is the fragmentation measured by a certain task m in the workload. We define the node-level measure as the weighted sum of the task-level, i.e.,

$$F_n(M) = \sum_{m \in M} p_m F_n(m). \quad (1)$$

One can interpret Eqn. (1) as *the expected fragmentation* measured by a task that is *randomly sampled* from the target workload. We next describe how $F_n(m)$ can be computed.

Pictorial Interpretation. From the view of a task, it considers a GPU of a node being fragmented if it cannot be allocated to the task. As a pictorial interpretation, we refer to Figure 4b and consider node A. In case that the node has multiple GPUs, we map their unallocated capacity (a vector) to a scalar representation following the approach described in §3.1. We partition the resource plane into four quadrants with node A at the origin. The node has insufficient resources to run any of the tasks that are located in Quadrants I, II, and IV, among which those in Q-I are in short of CPUs and GPUs (e.g., task D), Q-II in short of GPUs (e.g., task A), and Q-IV in short of CPUs (e.g., task C). From the point of these tasks, the unallocated GPUs are all fragmented, as they either have insufficient capacity (Q-I and Q-II) or become *stranded* due to the lack of CPU resource on the node (Q-IV).

Things become a bit more complex when it comes to Q-III. While the node has sufficient resources to run a task in that quadrant, not every unallocated GPU has enough capacity. For example, on a 4-GPU node with an unallocated capacity of $\langle 1, 1, 0.5, 0.25 \rangle$, the two partial GPUs cannot be assigned to a task that requests 2 GPU, even though the node has sufficient GPU resources. In this case, the two partial GPUs should be counted as being fragmented for the task. More generally, given a task in Q-III, we check each unallocated GPU, and those with insufficient capacity are considered fragmented.

In addition to the four quadrants, tasks can also locate on the x-axis if they request no GPU resource (e.g., task E). For these tasks, all the unallocated GPUs are considered fragmented as none of them can be utilized.

Formal Description. We now give a formal description of the computation of $F_n(m)$, where we consider only GPU and CPU resources. That being said, the fragmentation measure can be easily generalized to a high-dimensional space with more resources such as memory and network.

We start with a few notations. Given a node n with G_n GPUs, denote by $R_n = \langle R_n^{\text{CPU}}, R_{n,1}^{\text{GPU}}, \cdots, R_{n,G_n}^{\text{GPU}} \rangle$ the unallocated resource vector, where $0 \le R_{n,g}^{\text{GPU}} \le 1$ for all GPU g. Let R_n^{GPU} be the scalar representation of the unallocated GPU vector, which is defined as the number of fully unallocated GPUs plus the maximum partial GPU ($u = f + p$ in §3.1), i.e., $R_n^{\text{GPU}} = \sum_g \lfloor R_{n,g}^{\text{GPU}} \rfloor + \max_g (R_{n,g}^{\text{GPU}} - \lfloor R_{n,g}^{\text{GPU}} \rfloor)$. For each task m, denote by $D_m = \langle D_m^{\text{CPU}}, D_m^{\text{GPU}} \rangle$ the resource demand vector, where $D_m^{\text{GPU}} \in [0, 1) \cup \mathbb{Z}^+$. We compute $F_n(m)$ in the following three cases.

Case-1 (Q-I, Q-II, and Q-IV): Task m cannot run on node n due to the lack of CPU or GPU resources, i.e., $D_m^{\text{CPU}} > R_n^{\text{CPU}}$ or $D_m^{\text{GPU}} > R_n^{\text{GPU}}$. In this case, all the unallocated GPUs are considered as fragments by task m. We have

$$F_n(m) = \sum_{1 \le g \le G_n} R_{n,g}^{\text{GPU}}. \quad (2)$$

Case-2 (Q-III): Task m can run on node m and has a GPU request, i.e., $D_m^{\text{CPU}} \le R_n^{\text{CPU}}$ and $0 < D_m^{\text{GPU}} \le R_n^{\text{GPU}}$. In this case, we check each unallocated GPU and those with insufficient capacity are identified as fragment in the view of task m. Note that if m requests one or more GPUs, all partial GPUs cannot be allocated and are hence fragmented. We have

$$F_n(m) = \sum_{1 \le g \le G_n} R_{n,g}^{\text{GPU}} \mathbb{1}(R_{n,g}^{\text{GPU}} < \min\{D_m^{\text{GPU}}, 1\}), \quad (3)$$

where $\mathbb{1}(\cdot)$ is an indicator function that returns 1 if the given condition holds, and 0 otherwise.

Case-3 (x-axis): Task m requests no GPU, i.e., $D_m^{\text{GPU}} = 0$. In this case, all the unallocated GPUs are deemed fragments by task m, as none of them can be utilized. We have $F_n(m)$ computed the same way as in Eqn. (2).

In essence, $F_n(m)$ measures the amount of available GPUs on node n that cannot be allocated to task m. Taking the expectation of $F_n(m)$ with respect to the popularity distribution of tasks (see Eqn. (1)), we obtain $F_n(M)$, which measures the unallocated GPU resources on node n that are *expected to be fragmented* (hence wasted) from the viewpoints of the entire workload M.

Fragmentation Rate. Once $F_n(M)$ is obtained, we compute the node's *fragmentation rate* as the ratio between the amount of fragmented GPUs and the unallocated GPUs, i.e.,

$$f_n(M) = \frac{F_n(M)}{\sum_{1 \le g \le G_n} R_{n,g}^{\text{GPU}}}. \quad (4)$$

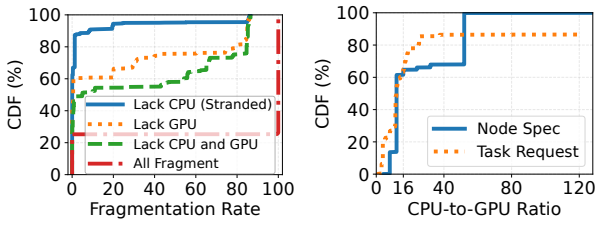
Intuitively, Eqn. (4) measures the severity of GPU fragmentation on a node.

Cluster-Level Fragmentation Measure. Given a cluster N and the target workload M, the cluster-level GPU fragmentation, denoted by $F_N(M)$, is the aggregate fragmentation of all nodes n in N, i.e.,

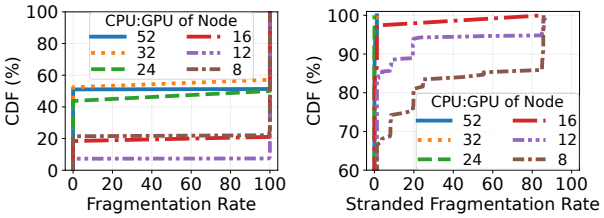
$$F_N(M) = \sum_{n \in N} F_n(M). \quad (5)$$

Normalizing $F_N(M)$ by the unallocated GPUs in the cluster gives the fragmentation rate of the cluster, i.e.,

$$f_N(M) = \frac{F_N(M)}{\sum_{n \in N} \sum_{1 \le g \le G_n} R_{n,g}^{\text{GPU}}}. \quad (6)$$



(a) CDF of node fragmentation rate. (b) CPU:GPU of nodes and tasks.



(c) Frag rate among various nodes. (d) Stranded frag rate among nodes.

Figure 6: Distribution of node fragmentation and CPU-to-GPU ratio of node and task in the fully packed cluster *H*.

3.3 Fragmentation Analysis in Action

The fragmentation measure described above can help operators better reason about the cluster state. To demonstrate its practical utility, we perform fragmentation analysis in various production clusters.

High Fragmentation Blocks Further Allocation. We reexamine a production cluster *H* previously described in §2.2. For each node, we measure its GPU fragmentation rate (Eqn. (4)) and depict the distribution in Figure 6a (red dash-dotted line). We observe that 25% of the nodes have GPUs fully allocated and are free of fragmentation, while the other 75% nodes measure over 99% fragmentation rate. This explains why the cluster cannot run any tasks even if it still has a capacity of 500 unallocated GPUs.

Breakdown Analysis of the Fragmentation Causes. Using the quadrant interpretation descried in §3.2, we can break down the fragmentation of a node into different causes: (1) having insufficient CPUs and GPUs to run tasks in Q-I (Figure 4b); (2) having insufficient GPUs to run tasks in Q-II, some GPU-sharing tasks in Q-III, and tasks that request different types of GPUs (not shown in Figure 4b); (3) having insufficient CPUs to run tasks in Q-IV (stranded GPUs); (4) running non-GPU tasks (x-axis) on a GPU node.

Figure 6a attributes the fragmentation rate to different causes and depicts their distributions. We see that the high fragmentation is primarily caused by the node lacking sufficient CPU and GPU resources (green dashed line), followed by lacking GPUs only (orange dotted line). This suggests that in cluster *H*, the allocation of CPUs and GPUs are relatively balanced. But still, a small number of nodes (4.6%) attribute stranded GPUs as the dominant factor (over 80%) of fragmentation (blue solid line).

Impact of CPU-to-GPU Ratio. In our analysis, we are interested in knowing which nodes are more likely to become fragmented and find the CPU-to-GPU ratio a good indicator. Figure 6b compares the CPU-to-GPU ratio of the node specs and the task requests. On one hand, 65% nodes (tasks) have (request) $\leq$ 16 CPUs per GPU, for which it is a good match between the node specs and workload demands. On the other hand, the cluster workload also contains 13% non-GPU tasks (with an infinite CPU-to-GPU ratio), and they account for 23% of all CPU requests. The existence of non-GPU tasks renders nodes with low CPU-to-GPU ratios more likely to become fragmented, especially with stranded GPUs.

Figure 6c correlates the fragmentation rate with the node's CPU-to-GPU ratio. Among the nodes with CPU-to-GPU ratio $\leq$ 16, 80% of them measure high fragmentation rate over 98% (dark red lines near the bottom). This proportion drops to only 47% when it comes to the nodes with 52 CPUs per GPU (light blue lines in the center). As for the fragmentation caused by stranded GPUs, Figure 6d shows that they are more commonly observed on the low-CPU nodes (e.g., 8 or 12 CPUs per GPU). We therefore recommend adding more high-CPU nodes to the cluster for reduced fragmentation and improved utilization.

Fragmentation in a Less-Crowded Cluster. Fragmentation is not a unique problem to fully packed clusters. Referring back to Figure 2, we measured the the cluster-wide fragmentation rate (Eqn. (5)) between 21% and 42% in a 1280-GPU cluster in a normal (off-peak) period with 77.6% average GPU allocation rate. Breakdown analysis further shows that 60% of the fragmentation was caused by GPU shortage. This highlights the importance of reducing fragmentation by packing existing GPU tasks, even in periods where the cluster is less crowded. Additionally, non-GPU tasks contributed only 13% to fragmentation in this cluster.

Running CPU Workloads in GPU Clusters. Our fragmentation analysis has also identified a pathological case. We have observed in a large cluster *B* consisting of 6.4k GPUs and 750k CPU cores, there were 84% of tasks requesting no GPU, resulting in many pending GPU tasks when the GPU and CPU allocation ratios reached only 80% and 86%, respectively. This led to the waste of 1.2k GPUs, with 95% of them considered being stranded. The main cause of this issue was *the lack of CPU reservation on GPU nodes* when scheduling non-GPU tasks. To address this problem, we recommend migrating non-GPU tasks to CPU nodes or even CPU clusters in order to decrease fragmentation and improve resource utilization. However, implementing this solution requires coordination and collaboration in areas such as scheduling, quota design, admission control, and capacity planning.

4 Fragmentation Gradient Descent

The fragmentation measure not only helps operators reason about the cluster state, but can also be used to guide task scheduling. In this section, we present the Fragmentation Gra-

dient Descent (FGD) algorithm that schedules tasks towards the direction of the steepest descent of fragmentation (§4.2), thereby achieving the maximum GPU allocation rate. We start with a description of the scheduling problem (§4.1).

4.1 Online Task Scheduling

We consider a GPU cluster managed by a container orchestration system such as Kubernetes [1] and Borg [30], in which tasks are submitted as pods and maintained in a queue. In its simplest form, tasks are scheduled in a first-come-first-served (FCFS) manner. For each task pod, the scheduler finds the best node and, if necessary, GPU(s) for it to run on. If no node can be assigned, the pod remains unscheduled and is pending for another scheduling attempt (e.g., placed to the end of the queue after a certain timeout). This formulates an *online scheduling problem*, in which tasks are revealed sequentially to the scheduler for placement decision making.

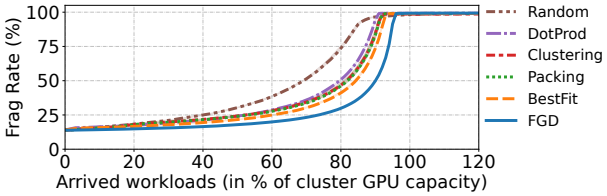
Fragmentation as a Metric. Each time a pod is assigned to a node, the remaining resources decrease, moving the node closer to the origin in the resource plane (Figure 4b). As a result, the fragmentation region (Q-I, Q-II, and Q-IV) expands and the fragmentation rate grows. Figure 7a depicts the growing fragmentation rate as the arriving tasks are scheduled under different policies in our trace-driven emulations (details in §6.1). Compared with the absolute measure (§3.1), the fragmentation rate is more sensitive to the placement decision and can be used as an indicator of the scheduling quality: higher fragmentation rate suggests a poorer scheduling decision.

Unlike the fragmentation rate, the fragmentation amount has no clear trend of growing or decreasing, as shown in Figure 7b. This is because by Eqn. (5), it is the product of the fragmentation rate and the unallocated GPUs – whilst the former grows as more tasks are scheduled, the latter decreases. Also note that the fragmentation starts with 13% of the total GPU capacity – all attributed to non-GPU tasks – but ends up at different degrees under different policies when the cluster is fully packed. At that point, all unallocated GPUs are fragmented (100% fragmentation rate in Figure 7a).

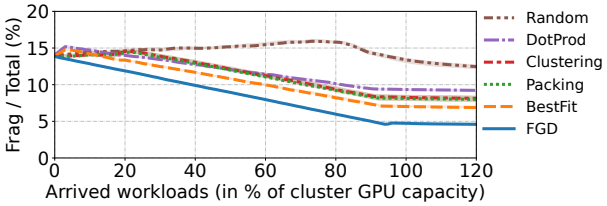
4.2 FGD Algorithm

Key Insight. From the previous discussions, we see that an effective approach to minimizing fragmentation is to *suppress the growth of fragmentation rate as much as possible*. This can be achieved by scheduling tasks towards the steepest descent of fragmentation, a heuristic called Fragmentation Gradient Descent or FGD.

Algorithm Description. Algorithm 1 formalizes the description of FGD scheduling. Given a task (pod) to schedule, the scheduler first filters out all the unavailable nodes with insufficient resources or unsatisfied placement constraints (lines 3–4), such as the lack of requested GPU types. The scheduler



(a) Fragmentation rate grows to 100% as more resources are allocated.



(b) Percentage of fragmented GPUs to total resources under our measure. It reveals the differences between various strategies from the early stage.

Figure 7: FGD pursues the lowest fragmentation among various policies in scheduling production workloads (more results under the same experiment settings are shown in Figure 9).

then *hypothetically* assigns the task to each node and calculates the increment (can be negative) of fragmentation that would be caused by each assignment. In case a task requests a partial GPU, the hypothetical assignment needs to try each GPU as well (line 6). Note that the hypothetical assignment can be performed *in parallel* for acceleration (lines 2–7). The scheduler finally assigns the task to the node (and the GPU) that causes the minimum increment of fragmentation (line 9).

Complexity and Scalability. FGD has a low computational complexity and scales to a large cluster. For a cluster with N nodes, the score of each node can be evaluated *in parallel* (Algorithm 1 line 2) and the scheduler simply selects the minimum. Besides, the scale of M is also limited since it only represents the number of *distinct* task resource requirements (e.g., 80 among 7.6k tasks). In our evaluation (§6), each scheduling decision can be made in *hundreds of milliseconds* on a cluster of $N = 1.2k$ nodes.

A Running Example. To better illustrate the scheduling process of FGD, we refer to Figure 8 for a running example. Consider a node with $\langle 0.5, 1\rangle$ unallocated GPUs. The target workload has three tasks with equal popularity, each requesting 0.3 GPUs (task-A), 0.5 GPUs (task-B) , and 0.7 GPUs (task-C). Originally, only GPU-A is considered fragmented (by task-C only). It thus measures the fragmentation of $0.5 \times 1/3 = 1/6$ GPUs. Assume that task-A arrives first. Assigning it to GPU-A increases the fragmentation to 0.2 GPUs whereas assigning it to GPU-B results in no increase. FGD hence assigns task-A to GPU-B, leaving the node with $\langle 0.5, 0.7\rangle$ unallocated GPUs. Next for task-B, FGD assigns it to GPU-A, reducing the fragmentation to 0. In comparison, assigning it to GPU-B would increase the fragmentation by 0.2 GPUs. Finally for task-C, it has no choice but to run on GPU-B, the only GPU with the sufficient capacity.

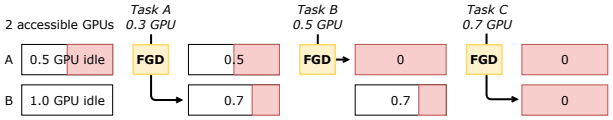


Figure 8: A running example of FGD scheduling. The target workload contains three tasks with equal popularity.

Algorithm 1: Task scheduling of FGD

Input : Cluster N, incoming task m, target workload M
Output : Assigned node n^*

Initialize node score set $\mathcal{S} \leftarrow \varnothing$, and output $n^* \leftarrow \varnothing$.
parallel for *node* $n \in N$ **do**
  if *Insufficient resources* $\|$ *constraints not met* **then**
    Return ▷ Filter out unavailable nodes
  $n^- \leftarrow \text{AssignTaskToNode}(m,n)$ ▷ Hypothetically
  $\Delta \leftarrow F_{n^-}(M) - F_n(M)$ ▷ Fragmentation increment
  $\mathcal{S} \leftarrow \mathcal{S} \cup (n,\Delta)$
if $\mathcal{S} \neq \varnothing$ **then**
  $n^* \leftarrow \arg\min_{n \in \mathcal{S}} \Delta$ ▷ pick the node with the least Δ.

As FGD considers both the resource availability and task distribution, it exhibits distinct behavior compared to other scheduling algorithms. Back to the provided example, for task-A scheduling, the best-fit (and similar bin packing policies) would prioritize GPU-A (0.5 GPUs idle) over GPU-B (1 GPU idle), whereas for task-B scheduling, the worst-fit (and other load-balancing policies) would choose GPU-B (0.7 GPUs idle) over GPU-A (0.5 GPUs idle). These policy preferences diverge from the scheduling choices made by FGD.

5 System Implementation

We have implemented a prototype scheduling system on top of Kubernetes v1.25.0 [1] in over 10k lines of Go codes. Our system consists of two main components: a standalone scheduler and an event-driven emulator.

A Standalone Scheduler. Kubernetes provides a pluggable architecture called the *scheduling framework* [2] for developers to implement a customized scheduler. Following this standard approach, we have implemented FGD and many other scheduling policies as individual score plugins. The scheduler listens for task creation events from the Kubernetes API server and maintains a queue to cache submitted tasks which are scheduled on a first-come-first-served basis.

Also, to enable fine-grained GPU allocation, which is not supported by native Kubernetes, we have implemented a *GPU-sharing* plugin to manage GPU resources. It filters out nodes with insufficient GPUs or mismatched GPU types, assigns tasks to the node with the *highest* scheduling score, and keeps track of the allocatable GPU resources on each node.

Event-Driven Emulator. The event-driven emulator interacts with the API server to manage the creation and deletion of nodes and tasks. It supports two modes: *high-fidelity simulation*, which can receive production traces as input and simulate the scheduling process in a large cluster consisting of tens of thousands of GPUs within a few hours; and *real deployment*, which can take over a production cluster with valid certificates and create task pods on real nodes. The goal of our system is to study the packing efficiency and resource fragmentation of different scheduling policies, which are critical in large clusters. Although it is not possible to perform experiments on real clusters with thousands of nodes, we use the *high-fidelity simulation* mode in our evaluation. This mode uses real traces as input and simulates task placements and resource allocations based on the scheduling logic, which yields the same scheduling results as real deployment.

6 Evaluation

In this section, we conduct extensive experiments to demonstrate the effectiveness of our scheduling policy FGD. We first compare FGD with several state-of-the-art mechanisms in scheduling production workloads (§6.2). Further, we examine the generality of FGD in various scenarios, covering a variety of traces featured by GPU-sharing (§6.3), multi-GPU (§6.4), GPU-type-constrained (§6.5), and non-GPU tasks (§6.6).

6.1 Methodology

Baselines. We compare FGD with five state-of-the-art heuristic policies for scheduling GPU workloads:

1. Best-fit (BestFit) [11,21,24] assigns tasks to the node with the least remaining resources, computed as the weighted sum of all resource dimensions.
2. Dot-product (DotProd) [8, 23, 24] allocates tasks to the node with the smallest dot-product value between the node's remaining resources and the task demands.
3. GPU Packing (Packing) [31] prioritizes task assignment to occupied GPUs, followed by idle GPUs on occupied nodes, and finally to fully idle nodes. The intuition is to reserve available resources for multi-GPUs tasks.
4. GPU Clustering (Clustering) [33] packs the tasks of the same GPU request together (GPU-sharing tasks are packed together). It avoids heterogeneous distribution of task resource requirements on the same node.
5. Random-fit (Random) distributes the task randomly to any node that meets the requirements for load balancing.

Since most of the policies above provide no native support of GPU-sharing workloads, we made some simple extensions as follows: 1) GPU resources of multi-GPU nodes are summed up as one dimension; 2) GPU-sharing tasks scheduled to a node are placed on the available GPU with the least remaining resources (i.e., BestFit); 3) multi-resource vectors are normalized by the maximum node capacity in the cluster.

GPU Request per Task	0	(0, 1)	1	2	4	8
Task Population (%)	13.3	37.8	48.0	0.2	0.2	0.5
Total GPU Reqs. (%)	0	28.5	64.2	0.5	1.0	5.8

Table 1: Distribution of tasks in the traces of cluster H.

Monte-Carlo Workload Inflation and Metrics. To assess the cluster's ability to accommodate workload under a given scheduling policy, we employ the Monte-Carlo *workload inflation* approach [29]. Specifically, we repeatedly submit tasks to the cluster for scheduling until they no longer fit. The tasks are randomly sampled from the traces with replacement [12], along with their resource requests and scheduling constraints. We repeatedly conduct each experiment 10 times and report the average and standard deviation of the key metric—the percentage of unallocated GPUs in the cluster when cumulative GPU requests reach 100% of the cluster capacity[5].

6.2 Allocation of Original Production Traces

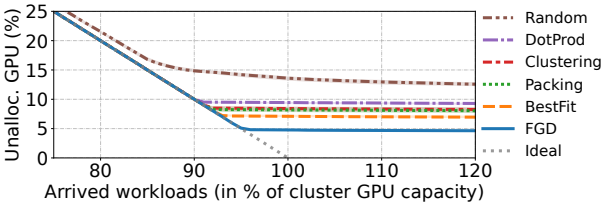
To evaluate the performance of different scheduling policies, we emulate the scheduling of over 8k tasks on the heterogeneous cluster H (§2.2) with 6.2k GPUs. As shown in Table 1, the majority tasks are the 1-GPU and GPU-sharing ones. Despite the small population of 8-GPU tasks, they still occupy a non-negligible portion in terms of requested GPUs (5.8%).

FGD Saves More Unallocatable GPUs. Figure 9a illustrates how the unallocated GPUs in the cluster decrease as tasks arrive under different scheduling policies. Ideally, the arrived tasks should be scheduled successfully until all GPUs are allocated (gray dotted line). However, in practice, the fragmentation rate also increases with the allocation of arrived tasks (Figure 7a). After a certain point when the fragmentation rate reaches 100%, all unallocated GPUs become fragmented and the allocation rate plateaus at a certain level, depending on the used scheduling policy (e.g., DotProd at 90%). Compared to classic scheduling policies, FGD achieves the highest allocation rate and reduces wasted GPUs by 33–49%, which translates to the additional allocation of 150–290 GPUs.

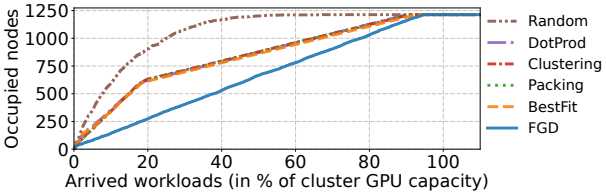
FGD Occupies Fewer Nodes in Scheduling. Figure 9b shows how the number of nodes that have at least one task running (occupied nodes) grows during the scheduling process under different policies. We observe that FGD packs tasks onto nodes whenever possible, leading to the fewest occupied nodes among all policies. Especially in early stages when the GPU allocation rate is 20%, FGD requires 55–70% fewer nodes to host all the tasks compared to other policies. As a result, FGD can enable significant savings in energy consumption (on premises) or operational cost (on clouds).

FGD Schedules More GPU-Sharing and One-GPU Tasks.

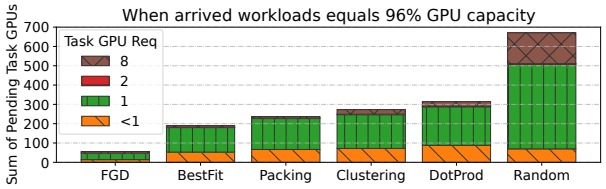
[5] Although a slight decrease in unallocated GPUs may still occur thereafter (thanks to few tiny tasks), the cluster is oversubscribed and rejects the majority of incoming tasks; this is not a desirable state to be considered in our metric.



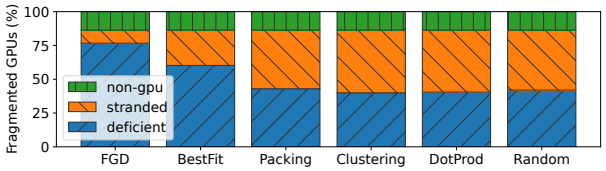
(a) The percentage of unallocated GPUs given arriving workloads.



(b) The number of GPU nodes occupied during the scheduling.



(c) GPU requests of failed tasks when the cluster is almost full (i.e., cumulative GPU requests reach 96% of the cluster capacity).



(d) The breakdown of GPU fragmentation into three causes.

Figure 9: Performance comparison of FGD and various scheduling policies. FGD outperforms all baselines with fewer unallocated GPUs and failed tasks.

Figure 9c depicts the distribution of GPU requests of unscheduled tasks under different policies when the cluster is almost full (i.e., the cumulative GPU requests of arrived tasks reach 96% of the cluster capacity). Except for Random, all policies schedule multi-GPUs tasks well. Compared to classic policies, FGD schedules up to 5.9× GPU-sharing tasks and 6.6× one-GPU tasks while reserving multi-GPU slots.

Early Detection of Fragmentation. Being able to detect fragmentation early on is the key to improving the GPU allocation rate. Referring back to Figure 7a, we see that FGD consistently achieves the lowest fragmentation rate among all policies, indicating better scheduling quality. As tasks arrives, the advantage of FGD becomes larger. When the cumulative requests grow to 90% of the total capacity, although most policies can still successfully schedule tasks, FGD outperforms alternative policies with 24–44% lower fragmentation rate. Figure 9d further decomposes the fragmented GPUs generated by each scheduling policy. It shows that $< 10\%$ of the fragmented GPUs by FGD come from stranded resources,

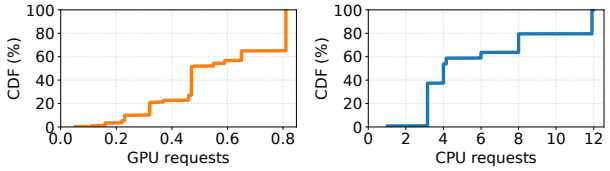


Figure 10: Distribution of resource requests of GPU-sharing tasks in cluster *H*.

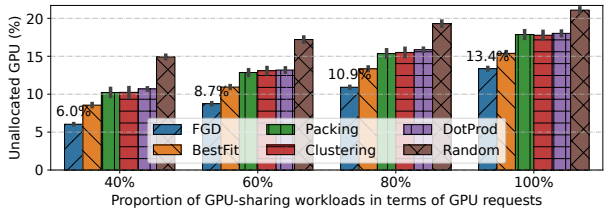


Figure 11: Scheduling results when the traces contain a varying number of GPU-sharing tasks. Y-axis shows the percentage of unallocated GPUs when the overall GPU requests of the arrived tasks reach 100% of the cluster's GPU capacity (same for the following figures).

which is 63–79% fewer than other policies. This confirms that reducing the amount of stranded resources provides better packing efficiency, which is in line with Borg's insights [30].

6.3 Allocation of More GPU-Sharing Tasks

GPU-sharing techniques enable finer-grained resource allocation. For GPU-sharing tasks, their resource requests are typically based on the actual usage of model execution. Figure 10 depicts the CDF of resource requests of GPU-sharing tasks. We observe that around 35% of GPU-sharing tasks request 0.8 GPUs, while no more than 5% tasks request less than 0.2 GPUs. This indicates that fractional GPUs (i.e., GPUs with partial resources allocated), after scheduling 0.8-GPU tasks, will mostly become fragmented. Nonetheless, there are still many packing combinations of tasks that can effectively maximize the use of shared GPUs. For example, 0.32-GPU tasks and 0.65-GPU tasks account for nearly 10% tasks, respectively—they can be placed together to reduce fragmentation.

FGD Tailors Resources for GPU-Sharing Tasks. To evaluate the impact of GPU-sharing tasks on scheduling, we construct different experimental settings based on production traces (§6.2), increasing the proportion of GPU-sharing tasks while keeping their resource requests in line with the original distribution. Fragmented GPUs increase as the proportion of GPU-sharing tasks rises (Figure 11). FGD considers the chances that remaining resources after packing can be utilized and tailors proper resources for subsequent tasks (Figure 8). It outperforms the other policies in all cases; even when the cluster has only GPU-sharing tasks, FGD reduces the unallocated resources of 125–290 GPUs.

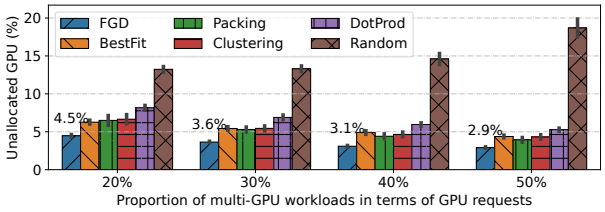


Figure 12: Various proportions of multi-GPU tasks.

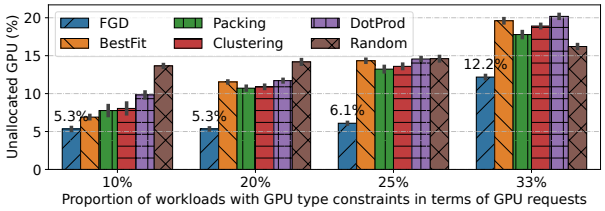


Figure 13: Proportions of task with GPU-type constraint.

6.4 Allocation of More Multi-GPUs Tasks

To better understand the scheduling impact of multi-GPUs tasks, we artificially adds more of them to the input workload, increasing their GPU requests from 20% to 50% of the overall demands. Figure 12 shows the scheduling results given by different policies. We observe that all policies except Random perform well when the workloads contain more multi-GPUs tasks. The reasons are two-fold: (1) As the population of multi-GPUs tasks increases, the scheduling impact caused by GPU-sharing tasks becomes less significant, making the classic multi-resource bin packing algorithms more effective. (2) Reserving multi-GPUs slot is desirable due to the presence of multi-GPU tasks. Compared to other scheduling policies, FGD avoids stranded resources and judiciously reserves GPU cards on nodes, reducing unallocated GPUs by 26–45% even when the multi-GPU tasks account for 50% of GPU requests.

6.5 Allocation of Tasks with GPU Constraints

In previous experiments, we have mainly considered fragmentation caused by *mismatches of resources*. Yet, production tasks also have placement constraints of desired GPU type. They typically request high-end GPUs for better performance; models are sometimes optimized for a specific generation of GPU. In our clusters, around 33% of GPU tasks have specified desired GPU types, while the rest can run on any GPUs. Heterogeneous GPUs in a cluster are often unevenly distributed, posing a challenge to resource scheduling.

Figure 13 shows the performance of different policies with varying numbers of tasks that specify GPU-type constraints. Classic heuristic policies only consider the alignment of resource demands, which is likely to cause severe fragmentation. In comparison, FGD reduces 32–40% of fragmented GPUs when tasks with GPU-type constraints account for 33% of the total GPU requests. FGD selectively reserves popular GPUs to avoid fragmentation caused by mismatches of GPU types.

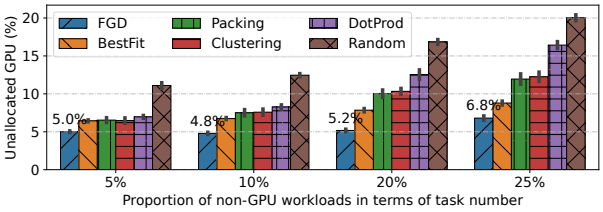


Figure 14: Various proportions of non-GPU tasks.

6.6 Allocation of More Non-GPU Tasks

In GPU clusters, there are often many tasks that request no GPU but other resources, such as CPU, memory, and disk. Examples include parameter servers and data processing tasks. These non-GPU tasks are usually not resource-intensive. Yet, if scheduled unwisely, they may cause some GPUs to become stranded. To evaluate their impact to scheduling, we vary the number of non-GPU tasks and compare the performance of different scheduling policies in Figure 14. FGD consistently maintains the unallocated GPUs at a low level, demonstrating its strong capability of avoiding stranded resources when making scheduling decisions. In fact, as the proportion of non-GPU tasks increases from 5% to 20%, fragmented GPUs caused by FGD increase by $< 3\%$ while other policies increase fragmentation by 18–45%.

7 Discussion

Scheduler Independence of Fragmentation Metrics. Our proposed fragmentation metric quantifies how fragmented the current cluster is considering only the next incoming task. This narrow focus ensures *scheduler independence*. Specifically, if we consider two or more incoming tasks, each node would need to determine the likelihood that the first task has consumed its resources when assessing the fragmentation measured by the second task. This determination would involve the scheduling policy. In contrast, our one-step fragmentation metric determines if a node is fragmented based solely on whether its remaining resources can be fully utilized by the next task, agnostic to the scheduler or other nodes.

Combining with Other Heuristics. Guided by the one-step fragmentation metric, schedulers may perform suboptimally in early stages when the cluster has abundant resources and little fragmentation is observed. However, the fragmentation-guided scheduling heuristic can be combined with other heuristics to navigate this initial period. For example, the scheduler could fall back to a best-fit approach if no increase in fragmentation is detected.

8 Related Work

Resource Fragmentation. Many research works address resource fragmentation in clusters [8,29,30,39,40]. For example, Tetris [8] shows that fair schedulers usually result in fragmented resources and delayed job completion, and proposes to combine them with an alignment-based policy. Borg [30] uses a hybrid scoring model to reduce the amount of stranded resources. HiveD [39] eliminates external fragmentation across multiple tenants by constructing virtual clusters. However, these works do not give a formal definition of fragmentation. We propose a statistical measure to quantify the degree of resource fragmentation and use it to guide task scheduling.

Multi-Resource Bin Packing. Resource allocation is often formulated as a multi-dimensional bin packing problem, which has been extensively studied [24]. Many heuristic policies, such as best-fit, vector alignment scoring (dot-product), are proven effective with high packing efficiency for scheduling big data analytics workloads [8] and virtual machines consolidation [11, 23]. However, it can be observed from our experiments that they do not work well in GPU sharing scenarios. Our work advocates resource allocation from the perspective of mitigating fragmentation, which can further reduce wasted resources.

GPU Cluster Scheduling. Recent works on GPU scheduling concern various objectives, such as cluster utilization [21, 32–34, 37, 40], job completion time [10, 18, 25, 26], job performance [22], and fairness [7, 20]. Although some research efforts (e.g., Gandiva [33], Salus [37], AntMan [34], TGS [32]) exploit GPU sharing techniques to improve resource *utilization*, they do not address GPU fragmentation. Our work complements them by reducing fragmentation and improving the GPU allocation rate.

9 Conclusion

In this paper, we have identified the significant fragmentation problem caused by GPU-sharing workloads that are increasingly deployed in production clusters. To address this problem, we have proposed a novel metric to statistically quantify the degree of GPU fragmentation caused by various sources. Based on this measure, we have proposed a simple, yet effective scheduling approach, named Fragmentation Gradient Descent (FGD), that schedules tasks towards the direction of the steepest descent of GPU fragmentation. Large-scale trace-driven emulations show that FGD substantially achieves higher GPU allocation rate compared to existing packing-based heuristics, saving hundreds of GPUs.

10 Acknowledgment

We thank our shepherd Feng Zhang and anonymous reviewers for their valuable comments. We also thank colleagues from Alibaba Group for their feedback and assistance in the early stage of this work. This work was supported in part by the Alibaba Innovative Research (AIR) Grant and RGC GRF Grants (#16213120 and #16202121). Qizhen Weng was supported in part by the Hong Kong PhD Fellowship Scheme.

References

[1] Kubernetes: Production-grade container orchestration. https://kubernetes.io, 2023.

[2] Kubernetes scheduling framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/, 2023.

[3] NVIDIA multi-instance GPU, seven independent instances in a single GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2023.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2018.

[5] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, and Chandramohan A Thekkath. A case for disaggregation of ml data processing. *arXiv preprint arXiv:2210.14826*, 2022.

[6] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proc. USENIX OSDI*, 2020.

[7] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proc. ACM EuroSys*, 2020.

[8] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proc. ACM SIGCOMM*, 2014.

[9] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. GaiaGPU: Sharing GPUs in container clouds. In *Proc. IEEE ISPA/IUCC/BDCloud/SocialCom/SustainCom*, 2018.

[10] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proc. USENIX NSDI*, 2019.

[11] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *Proc. USENIX OSDI*, 2020.

[12] John Hammersley. *Monte carlo methods*. Springer Science & Business Media, 2013.

[13] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proc. USENIX OSDI*, 2022.

[14] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proc. IEEE HPCA*, 2018.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE/CVF CVPR*, 2016.

[16] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proc. ACM/IEEE SC*, 2021.

[17] 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 *Proc. USENIX ATC*, 2019.

[18] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. AlloX: Compute allocation in hybrid clusters. In *Proc. ACM EuroSys*, 2020.

[19] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proc. ACM EuroSys*, 2023.

[20] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proc. USENIX NSDI*, 2020.

[21] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on multi-tenant clusters. In *Proc. USENIX OSDI*, 2022.

[22] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proc. USENIX OSDI*, 2020.

[23] Rina Panigrahy, Vijayan Prabhakaran, Kunal Talwar, Udi Wieder, and Rama Ramasubramanian. Validating heuristics for virtual machines consolidation. Technical report, Microsoft Research, 2011.

[24] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. Technical report, Microsoft Research, 2011.

[25] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic

resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*, 2018.

[26] 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 *Proc. USENIX OSDI*, 2021.

[27] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.*, 61(6):804–816, 2012.

[28] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads. *arXiv preprint arXiv:2202.07848*, 2022.

[29] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *Proc. IEEE CLUSTER*, 2014.

[30] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proc. ACM EuroSys*, 2015.

[31] 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 *Proc. USENIX NSDI*, 2022.

[32] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *Proc. USENIX NSDI*, 2023.

[33] 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 *Proc. USENIX OSDI*, 2018.

[34] 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 *Proc. USENIX OSDI*, 2020.

[35] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. KubeShare: A framework to manage GPUs as first-class and shared resources in container cloud. In *Proc. ACM HPDC*, 2020.

[36] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Xulong Tang, Chenchen Liu, and Xiang Chen. A survey of large-scale deep learning serving system optimization: Challenges and opportunities. *arXiv preprint arXiv:2111.14247*, 2021.

[37] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Proc. MLSys*, 2020.

[38] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient computation of the skyline cube. In *Proc. VLDB*, 2005.

[39] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C. M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *Proc. USENIX OSDI*, 2020.

[40] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proc. ACM SIGCOMM*, 2022.

Towards Iterative Relational Algebra on the GPU

Ahmedur Rahman Shovon[1], Thomas Gilray[1], Kristopher Micinski[2], and Sidharth Kumar[1]

[1]University of Alabama at Birmingham
[2]Syracuse University

Abstract

Iterative relational algebra (RA kernels in a fixed-point loop) enables bottom-up logic programming languages such as Datalog. Such declarative languages are attractive targets for high-performance implementations of relational data analytics in fields such as graph mining, program analysis, and social-media analytics. Language-level constructs are implemented via high-performance relational algebra primitives (e.g., projections, reorderings, and joins). Such primitives would appear a natural target for GPUs, obtaining high throughput on large datasets. However, state-of-the-art Datalog engines are still CPU-based, scaling best between 8–16 threads. While much has explored *standalone* RA operations on the GPU, relatively less work focuses on *iterative* RA, which exposes new challenges (e.g., deduplication and memory management). In this short paper, we present a GPU-based hash-join implementation, leveraging (a) a novel open-addressing-based hash table implementation, (b) operator fusing to optimize memory access and (c) two variant implementations of deduplication. To evaluate our work, we implement transitive closure using our hash-join-based CUDA library and compared its performance against cuDF (GPU-based) and Soufflé (CPU-based). We show favorable results against both, with gains up to 10.8× against cuDF and 3.9× against Soufflé.

1 Introduction

High-performance iterative relational algebra (RA) has the potential to automatically extract massive data-parallelism from applications built on top of bottom-up logic programming languages such as Datalog [2, 6, 9, 21, 38]. Datalog is a *declarative* logic programming language that has been applied to a wide variety of applications such as big-data analytics [17], graph mining [26, 33], and program analysis [5, 34]. The goal of declarative languages is for a user to provide a few understandable and compact rules that define a solution while the language automatically extracts an operational approach for computing the solution. Standard bottom-up Datalog solvers do just this: rules are implemented via standard RA operations and combined into kernels that infer new facts from discovered facts in a fixed-point loop.

Modern Datalog applications scale to extreme input-relation sizes (billions of rows/facts, tens of gigabytes of data), and thus highly parallel implementations are increasingly valuable. Unfortunately, however, modern CPU-based Datalog implementations have hit scalability walls due to their use of locking shared-memory data-structures—in our experiments, scalability for Soufflé (a best-in-class solver) peaks at roughly 16 threads. By contrast, modern GPUs offer (tens of) thousands of data-threads of computation in their high-throughput SIMD architecture. While there has been a plethora of work discussing implementation of standalone joins [20, 24, 30–32, 36] on the GPU, not much work tackles the problem of iterative joins, especially in the context of the modern GPU architectures. Iterative joins, fused with other relational operations such as union and projection, needed in the context of Datalog engines, add extra layers of complexity such as having to deal with low-level memory management, performing deduplication and mantainance for index data structures.

In this paper, we present key innovations which have allowed us to build scalable GPU-based implementations of Datalog-style declarative programs. Specifically, we implement the classic Datalog problem of finding all reachable paths (i.e., transitive closure) of a graph—a simple case of feature extraction. This problem entails implementing joins along with other RA operations like union and projection iteratively in a fixed-point loop. With this work, we make an important first step towards developing a complete GPU based datalog engine. Specifically, we make the following novel contributions to literature:

1. Developed a high-performance GPU-based hash table tailored to relational data; the hash table is used to accomplish binary hash joins between relations.
2. Implemented RA-operation fusion (e.g. join and projection) to improve memory and computation footprint.
3. Implemented deduplication using two techniques (a) sort

and unique (using thrust [4]), and (b) merge (two sorted lists) and unique. Both, key in facilitating iterative RA.

4. Evaluated our performance by comparing it against state of art openMP-based implementation, Soufflé and a cuDF implementation. We outperform both for almost all graphs and achieve speedups up to 3.9× over Soufflé and up to 10.8× over cuDF.

2 Related Work and Background

RA on GPUs Previous efforts in parallelizing RA have mostly focused on stand-alone implementations of select few RA primitives, such as join. Join is the most complex primitive to implement as its output size is not known in advance and depends on the characteristics of the input data and relations must be sorted, or stored in an index, for efficient joins to be possible. The most common algorithms for implementing joins are hash-joins and merge-sort join [3]. These algorithms have been extensively studied for shared memory systems, being parallelized for both GPUs [16, 19, 20, 24, 30–32, 36] and multi-core CPUs using openMP [23]. MPI-based distributed join operations have also shown promise in several recent studies [10, 13, 14, 25, 26]. However, unlike distributed join implementations, most extant GPU-based implementations do not maintain an order in their relations [39]. This poses a problem in some iterated relational algebra algorithms (such as transitive closure computation) as they require the join results to be sorted. If join results are sorted by default, we can avoid costly operations like sorting the entire result at each iteration of the algorithm, which impacts the overall performance of all iterations and compounds across the fixed-point loop. Additionally, off-the-shelf Python libraries also provide RA primitives to perform iterated join operations [35], but using the predefined methods does not allow fusing RA operations and creates memory overhead storing all intermediate results during iterated RA operations.

Datalog and iterated RA In Datalog, rules can be provided to define relations (tables) in terms of others. The following Datalog program inductively defines the transitive closure, T, of an input graph G using two rules: $T(x,y)$:– $G(x,y)$. and $T(x,z)$:– $T(x,y), G(y,z)$. The first rule represents a base case that says: every x-to-y edge in G implies an immediate x-to-y path in T. The second rule says: every x-to-y path in T that can be extended with a y-to-z edge in G, implies an extended x-to-z path. The second rule is recursive and must be iterated repeatedly until stabilizing at a value for T that is consistent with all rules.

The first rule can be implemented by inserting every element of G into T, a simple copy or union operation. The second rule can be implemented by iteration of a kernel function, composed of several RA operations, iterated to a least-fixed-point. One iteration of this function would join T on its second column with G on its first column, yielding all triples (x,y,z) where (x,y) can be drawn from T and (y,z) can be drawn from G. Projection to the set of unique (x,z) tuples, removing the middle column (as a graph, this is removing the intermediate vertex in the discovered path), and unioning this set of tuples with those in T completes one iteration of the second rule. The output (newly discovered facts) acts as the input for the following iteration, and the process continues till all reachable paths are discovered (see Figure 1).

3 Standalone Join

We first implement a standalone join operation between two input relations. A low-level programming model such as CUDA allows us to control the memory hierarchy and fuse operations together. We create a static hash table on the input relation and then use a hash-join-based approach to join the relations.

Representation We have developed a novel GPU-based hash table from scratch to meet requirements pertinent to our system. We extend the hash-join-based approach to support the relational data type specific to our application (2-ary column). This is done by implementing an efficient hash table that effectively supports search and can be easily linearized to a compact 1D array (required for deduplication)– both essential in implementing iterative joins.

To facilitate seamless hash-joins between relations, we use the entries of the join column of every relation as the key of the hash table. In particular, the join column of a relation is used as the *key*, which *maps* to a set of values (corresponding to the non-join column values). For example, for a relation that is hashed on the first column, with tuples $(1,2)$, $(1,3)$, and $(1,4)$ the key corresponds to 1 and the set of values would correspond to $2,3,4$. Note that, our hash-table uses only the join column as the key for hashing– making it easy to facilitate fast look-ups, needed for joins. Furthermore, to simplify our development (targeting graph analytic applications), we assume all base values are 32 bits wide.

All hash tables must support collision resolution. Two popular schemes include separate chaining and open addressing. However, GPUs face intrinsic difficulties in dealing with linked data, and we instead use an open-addressing-based hash table consisting of a fixed-size buffer. In our design, collision resolution is accomplished via linear probing, which (in our experiments) shows better cache performance than quadratic and double-hashing techniques. Our implementation uses the `Murmur3` hash, a popular implementation that performs well on open-source benchmarks [22]. The hash is calculated using a combination of bit shifts, multiplications, and XOR operations. We obtain a hash-table build rate of 4 billion keys per second for a string graph and 400 million keys per sec for a random graph (see Sec. 5 for more details).

Algorithm 1 Hashjoin based transitive closure computation algorithm. Blue boxes indicate join, orange indicates union, green indicates deduplication, and red indicates clearing.

```
1:  procedure TRANSITIVECLOSURE(Graph G)
2:      R ← HashTable(G)
3:      result ← Sort(G)
4:      T_Δ ← G
5:      repeat
6:          joinSizePerRow ← JoinSize(R, T_Δ)
7:          joinOffset ← Scan(joinSizePerRow)
8:          Initialize(joinResult, totalJoinSize)
9:          joinResult ← Join((R, T_Δ), joinOffset)
10:         joinResult ← Sort(joinResult)
11:         joinResult ← RemoveDuplicates(joinResult)
12:         totalUniqueJoinSize ← Size(joinResult)
13:         FreeMemory(T_Δ)
14:         T_Δ ← Copy(joinResult, totalUniqueJoinSize)
15:         unionSize ← resultSize + totalUniqueJoinSize
16:         Initialize(unionResult, unionSize)
17:         unionResult ← MergeSortedArrays(result, joinResult)
18:         unionResult ← RemoveDuplicates(unionResult)
19:         uniqueUnionSize ← Size(unionResult)
20:         oldUnionSize ← Size(result)
21:         FreeMemory(result)
22:         result ← Copy(unionResult, uniqueUnionSize)
23:         FreeMemory(joinOffset)
24:         FreeMemory(joinResult)
25:         FreeMemory(unionResult)
26:     until oldUnionSize ≠ uniqueUnionSize
27:     FreeMemory(R)
28:     FreeMemory(result)
29:     FreeMemory(T_Δ)
30:     return result
31: end procedure
```

In an open-addressing-based hash table, the load factor measures a table's sparsity, Load factor$(\alpha) = size(input)/size(hashTableSize) \leq 1$. Intuitively, a lower load will yield fewer collisions (and thus higher performance) [11]. In experiments, we use load factors 0.1 and 0.4 to pre-allocate our hash table as the nearest (larger) power of two (called *hashTableSize*); this permits us to replace the more expensive modulo operation in our hash function with an efficient binary AND operation. We create the hash table as a strided array of structures (*Entity*) with 32-bit integers *Key* and *Value* as fields. The hash table is initialized with a sentinel value (-1) to indicate an empty value. While doing bulk insertions in the hash table, CUDA threads search for the index of each tuple they are assigned, using the hash function. If the position is available (i.e., -1), we insert the tuple in the hash table using CUDA's atomic compare-and-swap operation (*atomicCAS*). If the position is already occupied, we search for the next available position using linear probing. In CUDA, if each thread only operates on one element of the input array, it can lead to problems if the input is larger than the number of available threads. To address this, we use a *grid-stride loop*, which allows a single thread to traverse multiple elements of the input array by incrementing its index by a stride value that is determined by the grid size and block size. Each CUDA thread performs build hash table computation on one grid size ($blockDim.x * gridDim.x$) at a time to provide maximum memory coalescing.

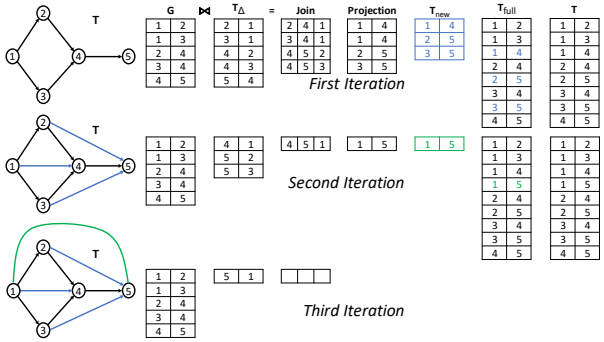


Figure 1: An example of the iterated joins on transitive closure computation of a graph

Hash-Join Phases We perform a join operation to get the join result of the hash table and input relation of *Entity* structure element with two 32-bit integers (*Key* and *Value* as structure members). We use two passes to compute the join result. These two passes are carried out by two CUDA kernels. First, we initialize an array (*offset*) with the size equal to the input relation. In the first phase, each CUDA threads searches for a total of n keys of the input relation in the hash table where n is the size of one grid($blockDim.x * gridDim.x$). If it finds a matching key in the hashtable, it increases the counter of the *offset* array for that input element. Thus, we calculate the number of matches for each key of the input relation using this CUDA kernel. We use thrust library's *reduce* and *exclusive_scan* APIs with device execution policy to get the total amount of matches and the exclusive prefix sum of the *offset* array. As the *offset* array reveals the size of the matches for each key of the input relation, the second pass inserts the joined columns to the join result array using another CUDA kernel. In the second pass, we do not insert the join column to fuse the join and projection operation in one kernel invocation. Both of these kernels use offset-based calculation without using any atomic operations or barrier synchronization, which are computationally costly. On top of that, our kernels use grid-stride loops to distribute the workload equally to the available CUDA threads. It also eliminates the branch divergence problem, where some CUDA threads could have more workloads than others.

4 Iterated Joins on the GPU

Implementing iterated joins requires the allocation of extra buffers to materialize intermediate results. Additionally, these intermediate results must be deduplicated between each RA iteration to ensure tractability. In this section, we demonstrate our approach to efficient iterated joins, using transitive closure as an illustrative example; our techniques generalize to other problems using finite-domain Datalog.

Transitive closure: Transitive closure is operationalized as iterated joins between a monotonically-growing set of transitive edges, T, and an extensional database of edges, G. With each iteration, new paths of increasing lengths are discovered (monotonically extending T) until all reachable paths are found and execution terminates. In practice, efficient implementations of Datalog employ semi-naïve evaluation [1], tracking a *frontier* of new facts rather than all-yet-seen facts. This is implemented by distinguishing multiple run-time tables for each syntactic relation: T_Δ, T_{full}, and T_{new}. T_Δ tracks the most-recently-discovered (last iteration's) transitive edges in T—T_Δ is merged into T_{full} after each iteration. Facts discovered during the *current* iteration are accumulated into T_{new}, which is swapped into T_Δ between iterations.

Figure 1 visualizes the execution of transitive closure on a line graph consisting of five nodes. As a preparatory step before the first iteration, T_Δ is populated by G. The first iteration joins G and T_Δ, yielding four (intermediate) triples. This is followed by projection of the join column, which results in a duplicate edge $(1,4)$. We deduplicate by sorting the results and applying consecutive deduplication (Thrust's unique, which required a preceding stable sort) to produce unique inferred paths in T_{new}. As a data-structure invariant, tuples are sorted (using the natural lexicographic sort): this enables merging T_{new} into T using a *single* scan of T_{new}. Next, we union the merged relation and generate the union result in T_{full}. This graph is visualized in the middle left side of Figure 1, where blue edges indicate the new unique paths found from the first iteration. We update the graph relation T_Δ with T_{new} for the next iteration. We continue this iterated join operations until we cannot find any new path or the size of T does not change after the deduplication of the merged relation.

Algorithm 1 formalizes our implementation of transitive closure in CUDA. This algorithm uses a combination of library functions illustrating the cruxes of Datalog on the GPU: joins (boxed in blue), union (orange), deduplication (green), and clearing (red). We begin by establishing the initial invariants: building an initial *result* by sorting G, and copying G into T_Δ. Next, we iterate until we reach a fixed-point; at the end of each iteration we compare the number facts in the next T with the number in the current T—when no new facts are added, the algorithm terminates at a fixed-point.

Join operation in transitive closure The join operation uses the single hash-join operation in a fixed-point iteration described in Section 3. We create the hash table (R in Algorithm 1) only once and repeatedly use it in the iterated hash join. The relation T_Δ is initialized with input relation G. The two-pass approach (Sec. 3) calculates the join result size (*joinOffset*) for each record of T_Δ and then inserts the join result in the *joinResult* array. Our algorithm fuses the join and projection operations together and thus reduces time and memory consumption for the iterated join (by up to 5%). This optimization is novel from an iterated join on the GPU perspective.

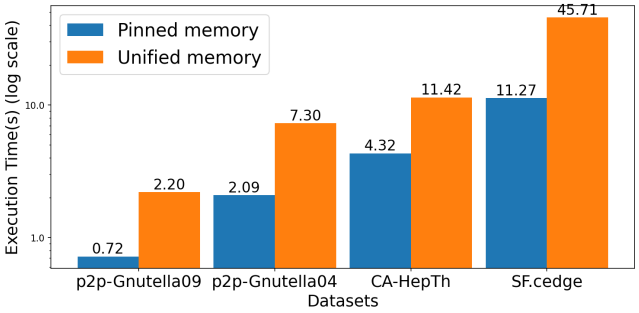


Figure 2: Time comparison between CUDA's pinned memory and unified memory model for hash join based TC computation. (log-scale for Y-axis)

Deduplication of the join result and merged result We use the key insight that some relational data can be permanently maintained in sorted order while some cannot and use this information to implement two kinds of deduplication techniques. Duplicate tuples are generated in two segments; after the join and projection operation and after the union operation. We use thrust library's *sort* and *unique* API to remove duplicates from the join result. The *sort* API first sorts the join results, and then the *unique* API removes the consecutive duplicate elements from the array. To deduplicate the union result, we initialize the *result* array with the input graph G and sort it. To remove duplicates from the union result ($result \cup T_\Delta$), we merge the sorted arrays ($result$, T_Δ). Then we apply the thrust's *unique* API to remove the consecutive duplicates from the merged result. This merging step eliminates the requirement of sorting the large *result* array before applying the deduplication process.

Memory management The CUDA programming model provides four memory allocation schemes: pageable memory, pinned memory, mapped memory, and unified memory [7]. Pageable memory involves two transfers: from host pageable memory to temporary pinned memory on the host, and then to device memory. Pinned memory initializes data on the host's pinned memory, requiring only one transfer to send it to device memory [18]. Mapped memory maps pinned memory in the device address space, avoiding host-to-device memory copying but increasing program processing time. Unified memory creates a managed memory pool for accessing data from both host and device using the same address, but introduces supplementary operations for memory management. For both the single join operations (Sec. 3) and iterated join operations (Sec. 4), we utilize the CUDA pinned memory model. We keep only two arrays (input relation and the final result) in the host memory, and all intermediate buffers are kept in the device memory without any need for data transfer

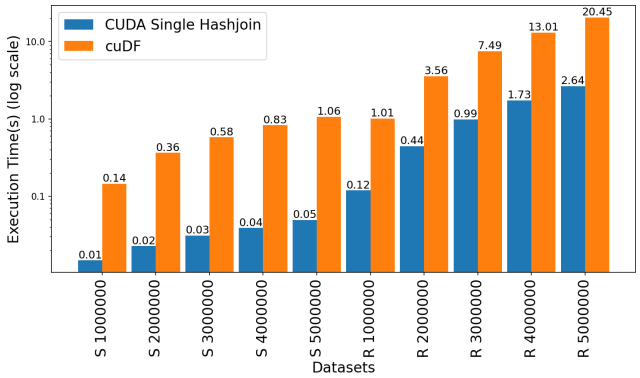


Figure 3: Time comparison between single join operation using CUDA and cuDF. R indicates Random, and S indicates String graph. (log-scale for Y-axis)

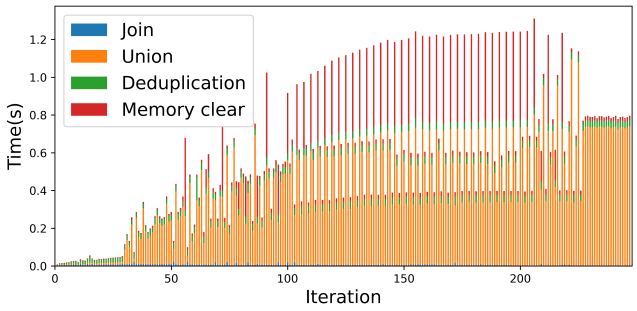


Figure 4: Our time breakdown for *fe_ocean* graph (CUDA outperforms Soufflé by 3.9×)

between the host and device memory for intermediate results. We ensure the implementations have no memory leakage using *cuda-memcheck* and *compute-sanitizer* API provided by CUDA library.

5 Evaluation

We perform a series of experiments to evaluate the performance of our iterative hash-based join implementation. We begin by evaluating the performance tradeoffs offered by using different memory management schemes of CUDA. Following which, we study the performance of a hash-join in isolation, comparing it against state-of-art join implementation of the cuDF library – part of NVIDIA's rapids framework that uses NVIDIA CUDA programming model for GPU parallelism [12, 15, 37]. Finally, we evaluate the performance of our iterative joins, by computing the transitive closure of a range of graphs. We compare our performance against Souffle, a state of art openMP-based library for performing iterated joins, and our cuDF-based implementation.

Experiment platform and datasets We conduct our experiments on the ThetaGPU supercomputer of Argonne National lab [27]. It has 24 NVIDIA DGX A100 nodes with eight NVIDIA A100 Tensor Core GPUs per node. Each node features two AMD EPYC 7742 processors with 3.31GHz clock speed and a total of 128 cores.

cuDF package was installed on a Python conda environment, and we developed our code using CUDA version 11.4. We use Souffle version 2.3 with 128 threads in ThetaGPU. As cuDF and our CUDA experiments use only one GPU device (single-gpu benchmark) we use a single GPU node from ThetaGPU. The single GPU node has 108 multiprocessors on device (SM) and we use $3,456$ (32×108) blocks per grid and 512 threads per block for each of the CUDA kernels.

To evaluate the experiments, we use real-world and synthetic graph datasets from the Stanford large network dataset collection, SuiteSparse matrix collection, and road network real datasets collection [8,28,29]. Table 1 shows all our graphs used along with performance results.

Memory optimization We evaluated the pinned memory, and unified memory as memory allocation schemes for transitive closure computation for several real-world graphs mentioned in Table 1 [7]. Figure 2 shows the time comparison between these two memory allocation schemes for directed and undirected graphs. We observe 2.6× to 4.1× speedup for transitive closure computation using the pinned memory over the unified memory model. However, we see that the unified memory model can handle graphs with larger TC size (e.g. dataset $p2p-Gnutella31$ from Table 1) without getting a run time error (out of memory error) as unified memory can oversubscribe the GPU global memory since CUDA version 8. Therefore, the graph $p2p-Gnutella31$, which has a higher workload per iteration overflows the GPU global memory (with pinned memory). For a single GPU-based implementation, this is an expected result.

Single hash-join performance To benchmark the single hash-join implementation (described in Sec. 3), we use two types of synthetic datasets; random graphs (numbers in the range of 0 to 32,767) and string graphs (e.g. $(1 \rightarrow 2), (2 \rightarrow 3), (3 \rightarrow 4)$). Both types of graphs have edges between 1 million to 5 million. Figure 3 shows the time performance comparison between cuDF and our standalone join.

For a random synthetic graph, we can build a hash table at a rate of 400 million keys per second. For the string graph, the build rate goes up to 4 billion keys per second. These two graph types invoke the two ends of the performance spectrum, with the random graph leading to tons of inefficient matches (worst case) and the string graph leading to perfectly aligned matches (best case). We observe up to 21× speedup for single hash-join computation for the string graph using our CUDA-based implementation over the cuDF join implementation. We notice that the increase in speedup is directly related to the size of the input relation for string graphs. For the random graph, we see 7.6× to 8.4× speedup using the CUDA-based

Table 1: Transitive closure performance using Hashjoin based CUDA, Souffle, and cuDF implementation. CUDA implementation uses 3,456 blocks and 512 threads per block. The Soufflé implementation uses 128 threads. Type *U* and *D* indicate undirected and directed graphs.

Dataset	Type	Rows	TC size	Iterations	CUDA Hashjoin(s)	Soufflé(s)	cuDF(s)
fe_ocean	U	409,593	1,669,750,513	247	138.237	536.233	Out of Memory
p2p-Gnutella31	D	147,892	884,179,859	31	Out of Memory	128.917	Out of memory
usroads	U	165,435	871,365,688	606	364.554	222.761	Out of Memory
fe_body	U	163,734	156,120,489	188	47.758	29.070	Out of Memory
loc-Brightkite	U	214,078	138,269,412	24	15.880	29.184	Out of Memory
SF.cedge	U	223,001	80,498,014	287	11.274	17.073	64.417
fe_sphere	U	49,152	78,557,912	188	13.159	20.008	80.077
CA-HepTh	D	51,971	74,619,885	18	4.318	15.206	26.115
p2p-Gnutella04	D	39,994	47,059,527	26	2.092	7.537	14.005
p2p-Gnutella09	D	26,013	21,402,960	20	0.720	3.094	3.906
wiki-Vote	D	103,689	11,947,132	10	1.137	3.172	6.841
cti	U	48,232	6,859,653	53	0.295	1.496	3.181
delaunay_n16	U	196,575	6,137,959	101	1.137	1.612	5.596
luxembourg_osm	U	119,666	5,022,084	426	1.322	2.548	8.194
ego-Facebook	U	88,234	2,508,102	17	0.544	0.606	3.719
cal.cedge	U	21,693	501,755	195	0.489	0.455	2.756
TG.cedge	U	23,874	481,121	58	0.198	0.219	0.857
wing	U	121,544	329,438	11	0.085	0.193	0.905
OL.cedge	U	7,035	146,120	64	0.148	0.181	0.523

implementation over the cuDF based implementation.

Iterated hash-join We compare our iterated hash-join-based transitive closure computation with state-of-the-art Souffle and Nvidia's cuDF library in Table 1. The benchmark results are based on multiple runs (at least 10), with the average being reported. The variance between runs was negligible (maximum standard deviation is <1%). For 15 out of 19 graphs, our CUDA-based implementation outperforms Souffle's implementation (128 threads). For the graph with the largest TC (*fe_ocean*, 1.6 billion edges), we observe a speedup of 3.9× over Souffle. We observe a speedup of 10.8× over cuDF implementation. Moreover, the cuDF implementation gets out of memory error several times where the CUDA based implementation is able to compute the transitive closure of those graphs using the same experimental setup. Additionally, we fused the projection operation with join operation in CUDA implementation, which is not possible in cuDF-based implementation. Thus, it shows both time and space performance enhancement of our iterated hash-join-based transitive closure computation. To better understand the time consumption for each of the individual operations (mainly join, union, deduplication, and memory clear) at the iteration level, we break down the operations at the granular level. One such granular benchmark is shown in Figure 4. We notice that the deduplication and union operation takes more time than the other operations. Also, as the number of join results is smaller in the first few iterations, it takes significantly less time in those iterations. For the graph (*usroads*), where we under-perform compared to souffle, our hypothesis is that we are not saturating the GPU as this graph has an increasing number of iterations (606) and less work per iteration.

6 Conclusion

We explored the issues for iterated operations such as inefficient operation fusion, GPU memory management, and facts deduplication while implementing the RA primitives which are necessary for developing Datalog applications. Our system is limited to a single GPU, and thus there are inherent scaling walls dictated by available VRAM on the GPU–by contrast, the largest unified nodes offer orders-of-magnitude more available RAM, supporting larger graphs. We view this work as a step towards multi-GPU joins across a cluster to develop a scalable backend for Datalog.

7 Acknowledgements

This work was funded in part by NSF RII Track-4 award 2132013, NSF collaborative research award 2217036, and NSF collaborative research award 2221811. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the ThetaGPU supercomputer located at the Argonne National Laboratory.

8 Availability

The data, code, and documentation are all open-sourced and can be found at https://github.com/harp-lab/usenixATC23.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[2] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1371–1382, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, January 2017.

[4] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2012.

[5] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, 2009.

[6] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[7] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[8] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[9] Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers. *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702. Springer, 2012.

[10] Ke Fan, Kristopher Micinski, Thomas Gilray, and Sidharth Kumar. Exploring mpi collective i/o and file-per-process i/o for checkpointing a logical inference task. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 965–972, 2021.

[11] David Farrell. A simple gpu hash table, Mar 2020.

[12] Alex Fender, Brad Rees, and Joe Eaton. Rapids cugraph. In *Massive Graph Analytics*, pages 483–493. Chapman and Hall/CRC, 2022.

[13] Thomas Gilray and Sidharth Kumar. Toward parallel cfa with datalog, mpi, and cuda. In *Scheme and Functional Programming Workshop*, 2017.

[14] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 23–35, 2021.

[15] Oded Green, Zhihui Du, Sanyamee Patel, Zehui Xie, Hang Liu, and David A Bader. Anti-section transitive closure. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 192–201. IEEE, 2021.

[16] Chengxin Guo and Hong Chen. In-memory join algorithms on gpus for large-data. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1060–1067, 2019.

[17] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, et al. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 881–884, 2014.

[18] Mark Harris. How to optimize data transfers in cuda c/c++, Dec 2012.

[19] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4), December 2009.

[20] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 511–524, New York, NY, USA, 2008. Association for Computing Machinery.

[21] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216, 2011.

[22] Mark Jarzynski and Marc Olano. Hash functions for gpu rendering. *UMBC Computer Science and Electrical Engineering Department*, 2020.

[23] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[24] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, page 55–62, New York, NY, USA, 2012. Association for Computing Machinery.

[25] Sidharth Kumar and Thomas Gilray. Distributed relational algebra at scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE*, volume 1, 2019.

[26] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In *International Conference on High Performance Computing*, pages 288–308. Springer, 2020.

[27] Argonne Leadership Computing Facility. Argonne leadership computing facility, 2022.

[28] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[29] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *International symposium on spatial and temporal databases*, pages 273–290. Springer, 2005.

[30] Ran Rui, Hao Li, and Yi-Cheng Tu. Join algorithms on gpus: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2541–2550, 2015.

[31] Ran Rui, Hao Li, and Yi-Cheng Tu. Efficient join algorithms for large database tables in a multi-gpu environment. *Proc. VLDB Endow.*, 14(4):708–720, December 2020.

[32] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Jiwon Seo, Stephen Guo, and Monica S Lam. Socialite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289. IEEE, 2013.

[34] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149, 2016.

[35] Ahmedur Rahman Shovon, Landon Richard Dyken, Oded Green, Thomas Gilray, and Sidharth Kumar. Accelerating datalog applications with cudf. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 41–45, 2022.

[36] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709, 2019.

[37] RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018.

[38] Jeffrey D Ullman. *Principles of database systems*. Galgotia publications, 1983.

[39] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 44:44–44:54, New York, NY, USA, 2014. ACM.

VectorVisor: A Binary Translation Scheme for Throughput-Oriented GPU Acceleration

Samuel Ginzburg Mohammad Shahrad[†] Michael J. Freedman

Princeton University [†]*University of British Columbia*

Abstract

Beyond conventional graphics applications, general-purpose GPU acceleration has had significant impact on machine learning and scientific computing workloads. Yet, it has failed to see widespread use for server-side applications, which we argue is because GPU programming models offer a level of abstraction that is either too low-level (e.g., OpenCL, CUDA) or too high-level (e.g., TensorFlow, Halide), depending on the language. Not all applications fit into either category, resulting in lost opportunities for GPU acceleration.

We introduce VectorVisor, a vectorized binary translator that enables new opportunities for GPU acceleration by introducing a novel programming model for GPUs. With VectorVisor, many copies of the same server-side application are run concurrently on the GPU, where VectorVisor mimics the abstractions provided by CPU threads. To achieve this goal, we demonstrate how to (i) provide cross-platform support for system calls and recursion using continuations and (ii) make full use of the excess register file capacity and high memory bandwidth of GPUs. We then demonstrate that our binary translator is able to transparently accelerate certain classes of compute-bound workloads, gaining significant improvements in throughput-per-dollar of up to 2.9× compared to Intel x86-64 VMs in the cloud, and in some cases match the throughput-per-dollar of native CUDA baselines.

1 Introduction

Server-side GPU acceleration has become ubiquitous, with all major cloud providers offering virtual machine instances with attached GPUs. GPU workloads such as graphics and machine learning have found widespread adoption due to the superior throughput-per-dollar that GPUs offer.

Typical approaches to accelerating these workloads on GPUs use domain-specific programming languages (DSLs). DSLs for GPUs heavily restrict which abstractions can be used by developers to write applications, and in particular forces them to use parallel abstractions. For example, machine learning programming systems such as TensorFlow [24] require users to specify programs as a series of operations performed on n-dimensional arrays. Approaches to extracting parallelism on GPUs for graphical workloads such as Halide [73] enforce more extreme restrictions such as requiring developers to express image operations as pure mathematical functions, defining the value of each function at each point. Other DSLs targeting batch dataflow workloads require developers to express their program using built-in parallel functions, which impose additional restrictions on application logic [75].

Developers who cannot express their application logic using these restricted abstractions are stuck manually rewriting applications in OpenCL or CUDA, which expose a low-level programming interface. Complex programs that use large pre-existing libraries, or where extracting parallelism is difficult, can be time-consuming to write and require drastic modifications to run using GPUs.

In this paper, we explore the feasibility of an alternative programming model for GPUs—where we take existing single-threaded programs and execute many copies of them using GPU threads. Each GPU thread corresponds to an emulated CPU thread, running a single instance of the program. Unlike prior approaches [37] which utilize interpretation, we translate the input program to native GPU code, substantially boosting performance while enabling a wider variety of target languages and runtimes. Unlike OpenCL or CUDA programs, we provide support for system calls and a CPU-like flat memory model. While less efficient than manual translation, this approach substantially reduces the barrier to accelerating throughput-oriented workloads using GPUs, ultimately improving the throughput and cost efficiency of applications that would otherwise run on CPUs.

Many applications written to run on CPUs are single-threaded programs, often implemented using high-level programming languages with large imported libraries. Without modification, these applications do not map cleanly to existing GPU programming models (e.g., those using language-level parallel functions such as in TensorFlow or Halide). Instead, these workloads process requests independently, with

no inter-request synchronization or communication. Examples of these workloads include cryptographic operations, image manipulation, and compression. These workloads are generally amenable to GPU acceleration [60, 67, 73], but are frequently run on CPUs instead.

Effectively enabling this programming model requires one to overcome several technical challenges in dealing with the substantial differences between GPUs and CPUs. These differences include how programs are executed—e.g., in which programs are run to completion without preemption—as well as a lack of support in GPUs for system calls. Further, failing to take differences in GPU memory hierarchies into account can result in an order of magnitude decrease in read and write performance. Prior approaches to running unmodified programs on GPUs suffer from poor performance due to the overheads of interpretation [37] as well as compatibility issues such as the lack of support for system calls [45].

To explore this unique programming model for GPUs, we built VectorVisor—a system which utilizes a vectorizing binary translator for GPUs. VectorVisor is designed to accelerate existing and unmodified programs that are designed to run on CPUs but can benefit from GPU acceleration. Target programs are automatically translated to run on GPUs efficiently, eliminating the need for complex manual translation. In particular, VectorVisor uses WebAssembly [54] as the intermediate binary format, which enables secure, fast, and efficient compilation for a wide range of applications.

We overcome the differences in program execution and memory hierarchy by translating WebAssembly programs to run directly on the GPU as opposed to using interpretation. We show that the remaining differences between CPUs and GPUs can be bridged with a combination of three techniques:

Continuations: CUDA and OpenCL do not provide support for preempting running applications in addition to lacking support for system calls. Without preemption, we cannot dispatch system calls, making it impossible to run complex and unmodified programs. To bypass this issue, we implement *continuations* for OpenCL C. Continuations are language-level primitives that allow us to save the program state at arbitrary locations, and then resume execution at a later time. Doing so allows us to pause and resume running GPU kernels, and to provide support for system calls. We also benefit from the portability of our approach—enabling VectorVisor to be run with multiple GPU vendors (e.g., NVIDIA, AMD).

WebAssembly: WebAssembly (WASM) binaries are designed with performance, portability, and security in mind. Many popular languages can compile to WASM (e.g., Rust, Go, C, C++, AssemblyScript, and more), making it an ideal intermediate format. WASM binaries are designed with runtime JIT compilation in mind, persisting vital information not present in x86 binaries. WASM semantics provide VectorVisor with memory alignment information, register allocation hints, type-checks on operations, and language-enforced structured control-flow. We heavily utilize this information to deal with challenges such as efficiently making use of the substantially larger per-thread [14] register space on GPUs—which is crucial for maximizing performance. Other important performance optimizations are also enabled through this compile-time information.

Memory Interleaving: GPUs organize threads in *warps*, or groups of threads. Each thread in a warp has a numerical index, and threads with adjacent indices must access adjacent bytes for optimal performance—so that memory accesses can be coalesced together. Coalesced memory accesses enable GPUs to maximize memory bandwidth usage at the cost of a more complex programming model. To bridge the differences between the GPU and CPU memory hierarchies, we automatically interleave the memory of each virtual machine running on the GPU to transparently coalesce all memory accesses.

We demonstrate VectorVisor's capabilities to accelerate several unmodified, third-party applications which use popular open-source libraries. We then evaluate VectorVisor's efficacy using nine benchmarks with throughput-per-dollar as our primary metric. Selected benchmarks include multiple classes of workloads, some of which reflect ideal applications of VectorVisor, with others reflecting the limitations of our programming model. Comparisons against native x86-64 and WebAssembly versions of each benchmark are provided, showing that VectorVisor can achieve superior throughput-per-dollar. We also provide native CUDA versions of two benchmarks to evaluate the efficacy of our translation. Our paper makes the following contributions:

1. We introduce a novel cross-platform approach to running lightweight virtual machines using GPUs, where VMs securely execute native code to maximize performance and support multiple high-level languages.

2. We show that support for system calls can be efficiently provided using continuations in addition to supporting recursion and indirect calls in OpenCL.

3. We demonstrate that we can emulate a flat memory model using an efficient memory interleave, enabling existing programs to leverage the high memory bandwidth of GPUs.

4. We explore the implications of batch size, latency, and throughput on VectorVisor's programming model and discuss which categories of workloads are optimal for it.

5. We discuss the limitations of our system and optimal GPU configurations for it.

2 Motivation and Challenges

The past several years have shown a large increase in the availability of cloud accessible GPUs. GPUs that cost thousands of dollars are now available at affordable prices per hour. Developers can quickly test if accelerating their program using

a GPU is cost-effective without large up-front investments in GPU hardware. However, despite having strong parallel processing power and cloud availability, GPUs are not often used for running high concurrency server-side applications.

Translating programs originally intended to execute on CPUs to run on GPUs is difficult due to the substantial differences between the execution models and memory hierarchies. Today's approaches to tackling these issues either require strong language-level restrictions with unintuitive stumbling blocks for developers, or slower automated approaches such as interpretation [37].

2.1 Execution Model Differences

Taking advantage of the throughput that GPUs offer requires using a different execution model than CPUs offer. GPUs feature restrictions on both the application runtimes and control flow that limit the set of possible workloads.

Runtime Limitations: CUDA and OpenCL are the two most popular compute APIs available for GPUs, and they share a near identical programming model. Programs that run on GPUs (GPU kernels) are submitted and execute until completion without preemption. High-level languages targeting general-purpose GPU programming such as CUDA C++ and OpenCL C feature restrictions on the usage of standard libraries, recursion, indirect function calls, variable length arrays, virtual functions, and templates [12, 16]. Support for other common features such as system calls and preemption are absent, further restricting the set of programs that can run.

Divergence: Unlike CPUs, which allow for different hardware threads to execute different instructions, GPUs organize threads into groups of threads (warps). Each warp shares a program counter, so all threads execute the same instruction on each clock cycle. Support for conditional branching is provided by executing no-ops for threads that have diverged, while the remaining threads block on threads executing the conditional branch. This results in the serialized execution of branches. Programs with substantial divergence are not able to efficiently use GPU resources as a result [33, 43, 51].

2.2 Memory Hierarchy Differences

GPUs and CPUs handle memory accesses differently due to the different design constraints imposed upon the hardware. CPUs optimize for reduced memory latency for all threads of execution, so they feature large cache sizes to minimize accesses to main memory. In contrast, GPUs seek to maximize memory bandwidth. GPUs can achieve $3\times$ the memory bandwidth of a comparable CPU [5, 8]. While GPUs can achieve higher memory bandwidth, memory latency on a given GPU can be up to $2.75\times$ worse than a comparable CPU [61, 66]. These differences in the memory hierarchy between GPUs and CPUs have two key implications for developers:

Register Space: At a high level, the memory hierarchies of CPUs and GPUs are similar, with both devices featuring registers, data caches, and byte-addressable memory. However, GPUs feature substantially larger register files. Each thread on recent NVIDIA GPUs can have a maximum of 255 32-bit register values [12] (just under 1 KiB of storage). In contrast, the x86-64 instruction set architecture has 16 64-bit general purpose registers (128 bytes of storage). Additionally, GPUs typically feature far more threads of execution than CPUs, further magnifying the difference. *Making use of this extra space is critical to maximizing performance on GPUs* [36, 70].

Memory Accesses: Rules regarding efficient memory access patterns are different for GPUs. GPUs require that programs perform *coalesced memory accesses*. Similar to CPUs, locality of memory accesses allows reads and writes to be cached, and is required to get optimal performance. However, GPU kernels must also ensure the locality of memory accesses across threads. Threads in a GPU warp are numerically indexed, and adjacent threads must access adjacent bytes of memory as a general rule. Otherwise, memory accesses within a warp can be serialized. Without proper memory coalescing GPU memory bandwidth can be cut by up to $32\times$ [12].

In addition to performance drops, GPUs can have stricter memory access policies than CPUs. Ideally, memory accesses are naturally aligned—meaning N byte accesses must be N-bytes aligned. Unaligned accesses cause running GPU kernels to fault on NVIDIA GPUs [12], causing programs that would have run correctly on a CPU to crash on a GPU.

3 System Design

In this section we introduce VectorVisor [1], a vectorizing binary translator for GPUs designed to leverage the implicit parallelism provided by our programming model. Our approach enables us to run many instances of unmodified programs on the GPU concurrently, making it far easier to utilize the substantial parallelism that GPUs offer.

We first describe our programming model in depth, where we explain how developers can leverage VectorVisor to accelerate programs. We characterize a set of ideal workloads for VectorVisor and explore the limitations of our programming model. Following this, we provide a high-level overview of VectorVisor's primary system components. To enable our simple and easy to use programming model, VectorVisor automates data transfer to and from the GPU. We illustrate this by showing the life cycle of a request processed by VectorVisor. Lastly, we provide an example of a short program, and how we transform it to run on the GPU.

3.1 Programming Model, Target Workloads

In contrast to OpenCL or CUDA, VectorVisor mimics the abstractions provided by CPU threads, treating each individual GPU thread as a small virtual machine. Each VM operates on a statically allocated chunk of memory, fully isolated from

[1] https://github.com/SamGinzburg/VectorVisor

other VMs. This memory model does not support inter-VM communication, and thus prevents deadlocking. Many copies of the same program are mapped to GPU threads, which then operate on distinct inputs.

This approach to parallelism for GPUs enables developers to run complex and *unmodified* single-threaded programs originally written for the CPU. Developers can leverage GPU acceleration without learning complex programming models or rethinking the logical structure of their programs.

VectorVisor is designed to function with unmodified workloads, but not all programs are equally amenable to acceleration using our programming model. Data parallel, latency-insensitive, and compute-bound 'serverless-like' workloads are ideal targets for GPU acceleration using VectorVisor. For a subset of these workloads, correct manual translation can be difficult without domain-specific knowledge—and those workloads represent an ideal use-case for VectorVisor. Suitable workloads share a number of characteristics:

High Execution Volume: VectorVisor relies on running many instances of the same program concurrently, instead of accelerating a single execution. Therefore, the more instances packed on the GPU, the higher the cost efficiency. Naturally, the latency QoS of the application should be able to afford the added batching latency prior to execution.

Application Limitations: VectorVisor runs unmodified programs where possible, but some abstractions are expensive to emulate on GPUs. Recursion and indirect calls reduce application performance due to how we implement them (explained further in Section 3.3).

Navigating tradeoffs between application concurrency and heap size are key to maximizing performance when using VectorVisor. We experimentally found in our evaluation that running 4096-6144 VMs with a heap size of 3-4 MiB proved optimal for our selected workloads. However, it is possible to run VectorVisor with varying degrees of concurrency—adjusting for different heap sizes.

Lastly, floating point differences between CPUs and GPUs can result in different outputs for applications [11, 37, 65], depending on the specific application and compiler.

Low Divergence: Ideal workloads should minimize program divergence in order to fully utilize the superior throughput and memory bandwidth that GPUs can offer.

Data Transfer Overheads: VectorVisor automates data transfer to and from the GPU; however, the overhead involved can be a substantial fraction of end-to-end request time. Maintaining a high ratio of GPU compute to input and output size is ideal for maximizing VM throughput.

3.2 Design Overview

VectorVisor consists of two key components, the binary translator (compiler) and the vectorized virtual machine monitor (VMM). We show an overview of VectorVisor in Figure 1, showing the role of each component as well as the life cycle of an incoming request to the system. Requests are first queued externally to VectorVisor, before being batched by the VMM, and submitted to the VMs running on the GPU—which are blocked on a system call awaiting input. We provide a pre-configured web server that automatically handles all data transfer to the VMM. After executing a batch of requests, responses are returned via another system call, and then back to the web server. This approach enables VectorVisor to be used as a drop-in replacement for existing systems, without the need for developers to manually batch incoming requests.

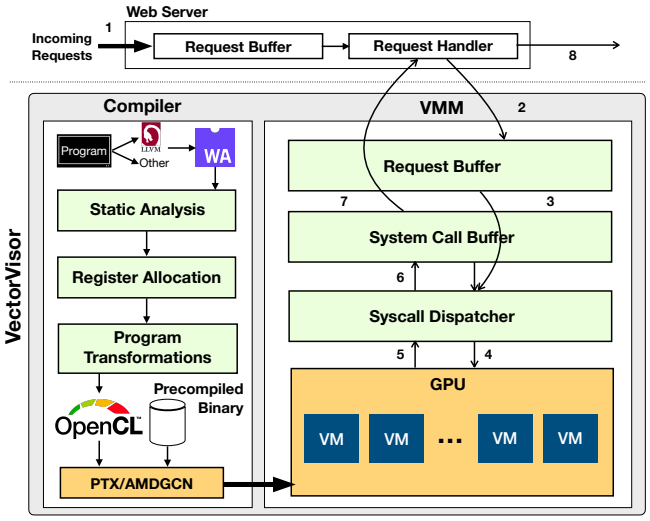


Figure 1: *System Overview*.

Binary translation is a separate process, occurring before applications run. Programs are compiled from any language targeting LLVM [64] (e.g., Rust, C, C++) into WebAssembly, our intermediate binary format. We then compile WebAssembly to OpenCL C. Targeting OpenCL C enables VectorVisor to support multiple GPU vendors. This approach allows us to run existing programs without the need to worry about complex language semantics—we only need to concern ourselves with WebAssembly semantics which are far simpler than alternatives such as LLVM IR and directly compiling high-level languages. LLVM IR places minimal restrictions on control flow structures, and can represent programs that are impossible for any GPU to run. WebAssembly only provides structured control flow by design, ensuring that programs can always be translated to run on the GPU [54]. Alternative approaches that directly compile high-level languages to run using GPUs require substantial engineering effort, and can run into compatibility issues [37].

Our system design features a number of novel contributions that we employ to bridge the substantial differences that exist between GPUs and CPUs described in Section 2. Our contributions succeed in bridging most of the gaps in capabilities between CPU and GPU runtimes. Recursive and indirect functions limit performance for some workloads but do not limit our functionality or correctness.

```
;; // Pseudocode in C:
;; int main(void) {
;;   // fd 1 == stdout
;;   char text[] = "ABCD\n";
;;   return write(1, text, strlen(text));
;; }
(module
  (import "wasi_unstable" "fd_write"
  (func $fd_write (param i32 i32 i32 i32) (result i32
      )))
  (memory (;0;) 1)
  (export "memory" (memory 0))
  (export "_start" (func $_start))
  (func $_start (result i32)
      i32.const 1  ;; stdout
      i32.const 0  ;; iovec ptr
      i32.const 2  ;; entries
      i32.const 24 ;; out bytes
      call $fd_write
  )
  (data (i32.const 0) "\10\00\00\00\02\00\00\00")
  (data (i32.const 8) "\12\00\00\00\02\00\00\00ABCD")
)
```

Figure 2: *WebAssembly Example*. We show an example of a simple program which makes a single system call.

3.3 Compiler

VectorVisor uses a binary translator (compiler) to translate input programs such that they can run on the GPU. The role of the compiler is to automate away the difficulties involved in writing programs for the GPU that are outlined in Section 2. We explore a set of techniques for enabling the execution of unmodified programs, which we demonstrate using a simple example of an input program.

3.3.1 Compiling WebAssembly

WebAssembly (WASM) [54], is a low-level language designed for performance, size, portability, and security. WASM binaries differ significantly from x86-64 binaries, as they are designed to be recompiled before runtime, retaining significant compilation information that can be used. Using WASM as an intermediate format simultaneously allows us to avoid dealing with the complex semantics of higher-level languages (e.g., Rust, C, C++) while also improving the performance of VectorVisor. We make use of this information in three places within our compiler:

1. Register Allocation: Recent NVIDIA GPUs have up to 255 32-bit registers per thread [12], providing roughly 8× the amount of storage per CPU thread ignoring vector registers. Traditional x86-64 binaries target CPUs with only 16 64-bit general purpose registers. Static analysis could conceivably be used to place stack allocations in x86-64 binaries into GPU registers, but WebAssembly provides a more convenient solution. In contrast to x86-64 binaries, WASM is a stack-based virtual machine and does not explicitly allocate registers [54]. Instead, values are placed either onto the stack or into local variables. Figure 2 shows an example WebAssembly program, which places four integers onto the stack. During compilation, we are able to store these values directly into variables which the backend GPU compiler (OpenCL C compiler) can then place into GPU registers. This approach allows the OpenCL C compiler to place values into registers that an x86-64 compiler would have placed on the stack.

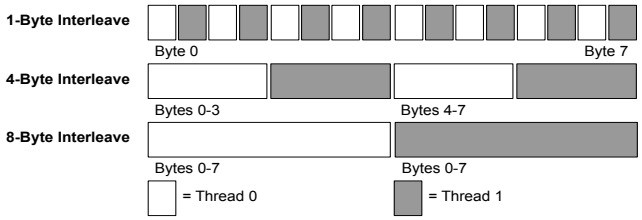


Figure 3: *Memory Interleaving* Examples of 1, 4, and 8-byte interleavings for a system with 2 threads are shown.

2. Runtime support: Most programs require some degree of modification to run on GPUs. Memory allocation, locking primitives, and threading primitives make assumptions about the underlying system that are false on GPUs. However, many such modifications are already performed by WebAssembly compilers. WASM binaries not only provide substantial compilation information, but also a "batteries-included" set of runtime modifications. Compilers targeting WebAssembly typically compile programs with a modified standard library with the necessary modifications already made.

3. Memory Alignment: Misaligned accesses cause running programs to crash when run using NVIDIA GPUs [12]. Handling misaligned accesses can be done at runtime by performing multiple aligned reads, but doing so introduces runtime overhead. Emitting optimized code for aligned accesses substantially boosts application performance. WASM binaries contain alignment information (e.g., the `align` attribute) that we can use to optimize reads and writes. However, the align attribute is only a hint, and as per the WASM specification, programs are expected to run correctly even with incorrectly specified alignments [54]. In practice, WASM binaries compiled by LLVM always contain the correct alignment information. By restricting the set of programs that we run to those compiled by LLVM, we can leverage these compilation hints safely to improve the performance of VectorVisor. VectorVisor supports running programs with and without this optimization using compiler-flags.

3.3.2 Memory Interleaving

GPUs have strict memory access rules to obtain optimal performance. As described in Section 2.2, GPU kernels must coalesce memory accesses to maximize memory bandwidth. Doing so requires developers to interleave objects in memory, such that adjacent threads access adjacent bytes, breaking the abstraction of a flat memory model. Other aspects of the flat memory model, such as process (or VM) memory isolation are also absent on GPUs by design.

VectorVisor provides the abstraction of a flat memory model to developers, automatically interleaving the address space of underlying virtual machines (threads) on the GPU

```
__kernel  void wasm_entry(...) {
  // Set up the stack, heap, buffers, ...
  do {
    /* call the next func/continuation */
    switch (*entry_point) {
      case 0:
        ___start(...);
        break;
      default:
        return;
    }
  // Check if we are done executing
  } while(*sfp!=0 && *syscall_number==-2);
}
```

Figure 4: The *trampoline* function serves as the entry point to each GPU kernel.

to provide both performance and security. This approach allows existing programs to run, while also extracting the full performance benefits of a GPU—assuming that running VMs exhibit similar memory access patterns. Randomized memory access patterns or significant program divergence can reduce memory bandwidth. Figure 3 shows how memory is interleaved across VMs in VectorVisor. Memory is organized into *cells* of contiguous bytes. Cell addresses are computed using the following pointer arithmetic (C operator precedence):

$$cell\_addr = (\frac{offset}{ileave}) \times (num\_vms \times ileave) + (vm\_idx \times ileave) + mem\_base$$

Where the interleave (*ileave*) represents the byte-width of the interleaving (e.g., 1, 4 or 8), the *offset* is the zero-indexed WebAssembly address, and *mem_base* is the base address of the allocated chunk of memory. Memory accesses are rewritten to operate on cells, with misaligned and larger (e.g., 8, 16-byte value) accesses requiring multiple operations. Our approach enables us to support 1, 4, and 8-byte interleavings, with larger interleavings typically achieving superior memory bandwidth.

WASM memory is represented as a zero-indexed linear array of bytes with pointers in the range of 0–2^{32}-1 and does not expose virtual addresses to running VMs. The relative addressing model WASM uses enables the compiler to control the virtual addresses of all memory reads and writes. Our cell address computation prevents VMs from computing cell adresses which belong to other VMs— preventing out-of-bounds accesses from corrupting or leaking data and providing memory isolation by construction.

3.3.3 GPU Preemption

Section 2.1 described the limitations of GPU programming models such as OpenCL and CUDA. Common features of programs such as system calls, recursion, and indirect calls vary in support—with system calls being absent from both OpenCL and CUDA. To fully mimic the execution environment provided by a CPU in VectorVisor, we support all three features. Implementing these features within OpenCL C requires us to provide support for preempting running programs. We provide support for preemption in VectorVisor by extending OpenCL C with support for continuations. Continuations provide the abstraction of being able to pause and resume programs at arbitrary points. To maximize the performance of VectorVisor, we leverage several compiler optimizations to reduce the overhead they introduce.

```
# 'c' is the called continuation
def example_fn(c, ...):
# context restore handler
switch (context):
  case 0:
    goto resume0;
  case n:
    goto resume_N;
  default:
    # Direct call, start from the top
# Indirect function call
switch(func_ptr):
  case 0:
    return c(example_fn, 0, resume0, ...)
    resume0:
  case n:
    return c(example_fn, 1, resume1, ...)
    resume1:
  default:
    trap;
# Optimized calls can be issued directly
call_example_func(...)
# Standard function call
return c(example_fn, 2, resume2, ...)
resume2:
# Recursive function call]
# In this case example_fn == 3
return c(example_fn, 3, resume3, ...)
resume3:
# System call
# These functions return control to the VMM
return c(example_fn, 4, resume4, ...)
resume4:
```

Figure 5: *Transformed Program* Pseudocode example of how different types function calls are implemented.

Continuations. Continuation-Passing Style [81] (CPS) is a relatively uncommon programming style where functions take in an additional parameter (the continuation), and instead of returning a value call the provided continuation with the return value. CPS with trampolining [27] is similar to standard CPS, with the difference being that function calls return continuations instead of just calling the provided continuation. A control operator (trampoline function), is used to repeatedly call the returned continuations. Figure 4 shows the trampoline function used in VectorVisor, which is the main entry point to each running GPU kernel. Implementing CPS with trampolines in this manner enables VectorVisor to preempt running GPU kernels at arbitrary locations—although we only return control to the CPU when either every VM is finished executing or when every VM is blocked on a system call. In Figure 5 we see that the only difference between recursive, indirect, and standard calls is the returned continuation (which encapsulates the program control state). This approach makes it easy to bypass OpenCL C language-level restrictions and provide support for recursive and indirect calls.

Compiler Optimizations. Naively implementing CPS with trampolines enables support for system calls and recursion

with large runtime overheads. To obtain better performance, VectorVisor performs static analysis to minimize the size of saved program contexts. We apply liveness analysis in addition to leveraging WASM type and control flow information to enable (1) incremental context saving, (2) loop-invariant code motion, and (3) WebAssembly-specific optimizations.

Liveness is associated with local usage inside WASM stack frames, and we insert all context save and restore operations around control flow instructions (e.g., `block`, `loop`, `br`, `br_if`, and `end`) and function (or system) calls. Runtime taint tracking is used to further enhance our liveness estimates.

Stack frame contexts are saved incrementally—only saving values written to since the previous context save operation. Liveness estimates are used to minimize context sizes in addition to only restoring live values when resuming continuations or unwinding stack frames. Loops without recursive or indirect calls can be further optimized—with context saving and restoring operations hoisted out of the loop. WASM function type signatures are used to translate amenable indirect calls into direct calls by filtering possible indirect call targets.

3.3.4 Profile-Guided Optimization

Minimizing the overhead of translating recursive and indirect calls is key to running complex applications. Compiler optimizations eliminate much of the overhead in the common-case. Edge cases, such as heavy usage of indirect and recursive calls in a tight loop remain a challenge. While recursion often cannot be eliminated without restructuring programs, indirect calls are easier to remove [25, 34]. Most indirect calls in high-level languages have only one target—with on average 73.5% of indirect call sites in Java programs being monomorphic [59]. Despite aggressive monomorphization in the Rust compiler [26], up to 37% of the most popular Rust libraries reduce code size by not removing optimizable indirect calls where possible [85]. Up to 98% of indirect calls in Java programs can be optimized out entirely [59].

We package a separate tool for instrumenting binaries, to implement profile-guided optimization for VectorVisor. Each program is instrumented and run using sample inputs representative of the overall workload. Using profiler data, we replace all indirect calls with less than 15 seen call targets with direct calls. To avoid emitting indirect calls to handle unseen targets, we instead emit panic handlers which check for valid targets.

3.3.5 Soundness

VectorVisor performs a 1-to-1 translation for all operations in input WebAssembly programs (e.g., stack operations, memory access, arithmetic, control flow). Limitations on the soundness of our approach come from (1) Compilation to WebAssembly and (2) Optimizations.

Most common workloads can be recompiled to WebAssembly without problems, but programs which rely on specific x86 instructions (e.g., 80-bit floats), language implementation details (e.g., undefined and implementation defined behavior), and complex language runtimes with unimplemented features (e.g., Go) can experience correctness issues.

Compiler-flags and tools (e.g., `wasm-snip`) are used to replace panic-related functions with `unreachable` statements. Unrecoverable errors can be expensive to handle, and in most cases replacing them with program aborts has no impact on correctness. Profile-guided optimization (PGO) can reduce indirect call counts, significantly improving performance in some cases. Our implementation of PGO only includes function calls we observe as potential targets at indirect call sites, aborting on unseen call targets. In practice, the indirect call targets we observed did not vary significantly with user-input beyond what we observed during profiling.

3.4 VMM

VectorVisor's VMM handles all data transfer between the running VMs on the GPU and CPU, as well as executing all system calls. The VMM greatly simplifies the use of VectorVisor by developers, avoiding the need to manage data transfer manually or to batch incoming requests.

Support for dispatching system calls is provided through the WebAssembly System Interface (WASI). We implement two custom WASI system calls—which are used to create a serverless-like event handler API for running VMs. Other implemented calls are primarily used to initialize language runtimes (e.g., reading environmental variables), support random number generation, serve as synchronization barriers (e.g., block on a subset of VMs), and perform simple IO (e.g., error logging).

Incoming requests are buffered using the request buffer, while system calls use an alternate buffer, as shown in Figure 1. Double buffering adds some overhead, but enables VectorVisor to overlap expensive network IO with on-GPU execution time. Sufficiently compute-intensive workloads prevent workloads from bottlenecking on the VMM, which can process thousands of VMs per-CPU. VectorVisor supports using pinned memory transfers with multiple GPU vendors (e.g., NVIDIA, AMD) to further optimize data transfer speeds—with vendor-specific optimizations [1, 4].

4 Evaluation

In this section, we present an evaluation of VectorVisor. First, we discuss the efficiency of (1) our memory interleaving and (2) system call implementation. Second, we explore a variety of modified and unmodified workloads to better understand the tradeoff space of our novel approach to accelerating programs. In several cases we show that we obtain superior throughput-per-dollar against x86 CPUs. Breakdowns of the end-to-end latencies of each benchmark are provided as well to explain our results. Finally, we evaluate the efficiency of our translation against handwritten CUDA baselines.

Instance Name	CPU	GPU	Cost/Hr
g4ad.xlarge	Intel Cascade Lake	AMD Radeon Pro V520	$0.3785
g4dn.xlarge	Intel Cascade Lake	NVIDIA T4	$0.526
g4dn.2xlarge	Intel Cascade Lake	NVIDIA T4	$0.752
g5.xlarge	AMD EPYC 7002	NVIDIA A10G	$1.006
g5.2xlarge	AMD EPYC 7002	NVIDIA A10G	$1.212
c5.xlarge	Intel Cascade Lake	N/A	$0.17
c5a.xlarge	AMD EPYC 7002	N/A	$0.154

Table 1: *Hardware Configurations.* Prices as of 1/5/2023.

4.1 Methodology

Testbed. We evaluated VectorVisor using Amazon Web Services (AWS). Five different VM types were used to compare against x86-64 baselines, and an additional two larger instances types were used to compare VectorVisor against CUDA baselines. We provisioned three VMs with attached GPUs (g4ad.xlarge, g4dn.xlarge, g5.xlarge), each with 4 vCPUs and 16 GiB of memory. Two additional compute-optimized VMs were used for evaluating CPU performance (c5.xlarge, c5a.xlarge), each with 8 GiB of memory and 4 vCPUs. Lastly, we used a single invoker VM (c5.8xlarge) for sending requests. These instances were used to obtain the results in Figures 8 and 9. Double extra large (2xl) instances have 2× the memory and CPU of smaller (xlarge) instances. These instances were used (in addition to xlarge instances) to evaluate handwritten CUDA programs. CUDA results which use 2xl instances can be found in Table 4. All VMs are allocated in `us-east-1`, in the same availability zone. Benchmarks are evaluated end-to-end over the network with IO and system overheads included in all measurements.

Some hardware configurations could not be evaluated due to AMD-specific bugs. AMD v520 GPUs, which are the only cloud-available AMD GPU on AWS, are unsupported in ROCm [21, 23] resulting in runtime crashes. Two benchmarks (Strings-Go and Strings-AScript) are built with WebAssembly-focused runtimes and do not have evaluated x86-64 configurations. Detailed results for all system configurations can be found in Tables 5 and 6 (Appendix).

Table 1 shows the hardware each VM has attached. NVIDIA configurations use the latest CUDA 12 backend, while AMD GPUs use ROCm 5.4.0 with the latest AMDGPU-Pro driver. For our GPU instances, we do not run any fraction of our workload on the available CPU cores to focus on evaluating GPU performance using VectorVisor. Undoubtedly, a hybrid CPU/GPU deployment would be more cost-efficient. We leave the exploration of heterogeneous deployments to future work. VectorVisor runs each benchmark using a 4- or 8-byte memory interleaving.

Workloads. Workloads are run using a cross-platform 'serverless-like' event-loop. Figures 11, 12, and 13 (Appendix) demonstrate how programs are written to run using our event-loop with x86-64, WASM, and VectorVisor. Our benchmarks evaluate VectorVisor using 'as-is' open-source library code in addition to optimized code written to run more efficiently. We explore the translation of lightweight (e.g., Rust) and complex language runtimes featuring garbage collection such as Go using TinyGo [22], and AssemblyScript [19]—a TypeScript-like language. Our x86-64 baselines are compiled at -O3 with SIMD enabled. Two benchmarks (Blur-Bmp, PHash-Modified) were manually rewritten to run natively using NVIDIA CUDA APIs to explore the efficiency of our translation. Table 2 shows each benchmark, its category, whether it features code that runs inefficiently in our system, whether we had to modify the imported library, batch sizes, and the total number of downloads on crates.io, a public repository for Rust libraries [10]. Benchmarks are run for 10 minutes to compute the average throughput (RPS).

Example Functions. Perceptual hashing is widely used in industry, such as by Facebook [6, 7], to cross-reference a given image against a database of images. We evaluate an open-source implementation of Blockhash—a variation on existing perceptual hashing algorithms [9, 90]. To further evaluate the efficiency of our translation, we also evaluate a modified blockhash library that we optimized to run more efficiently using VectorVisor. Additionally, we evaluate a bill generator which generates PDFs containing a set of purchased items formatted with a default template. Both benchmarks use mock data to simulate realistic workloads, with the hashing benchmark using 200×200 randomly generated images and the bill generation benchmark using 25 randomly generated item names and prices with an attached image.

Microbenchmarks. We evaluate a set of common microbenchmarks, including image processing workloads (e.g., Gaussian image blur), cryptography (e.g., password-based key derivation functions such as Scrypt and Pbkdf2), string compression (LZ4), histogram computation (Histogram), and string processing (e.g., stop word filtering and hashtag extraction). Our image processing, histogram, and cryptographic benchmarks operate on realistic, synthetic, and random inputs. We use the same parameters as Cisco type 8 passwords [20] for Pbkdf2 and Litecoin parameters for Scrypt [15]. To simulate realistic workloads, the compression and string benchmarks use as input a public dataset of tweets [41].

Baseline comparison. For our evaluation, we use two different baselines as points of comparison. First, for each of our benchmarks we compile them to WebAssembly (WASM), optimize them using `wasm-snip` and `wasm-opt` [17, 91], and execute them using Wasmtime [18] (a popular WASM JIT compiler). VectorVisor takes in the same WASM binary as an input. Second, for each of our benchmarks we compile and run them natively on an x86-64 CPU. This is the default choice for many developers who choose to run applications in the cloud, as most programs target x86-64. Each CPU benchmark is evaluated with multiple threads executing in parallel—proportional to the number of cores available.

Features	Scrypt	Pbkdf2	Blur Jpeg	Blur Bmp	PHash	PHash Modified	Bill PDF	Histogram	LZ4	Strings
Category	Crypto.	Crypto.	Image Proc.	Image Proc.	Image Proc.	Image Proc.	Misc.	Misc.	Misc.	Misc.
Reason for Inclusion	M	A	M & A	Alg.	Uses Indirect Calls	Alg.	D	A	A	D
Recursive or Indirect Code	×	×	✓	×	✓	×	✓	×	×	×
Unmodified Library Code	✓	✓	✓	✓	✓	×	✓*	✓	✓	✓
Batch Size (V520, T4, A10G)	2048, 4096, 6144	2048, 4096, 6144	1536, 3072, 4096	1536, 3072, 4096	1536, 3072, 4096	1536, 3072, 4096	1536, 3072, 4096	2048, 4096, 5120	1536, 3072, 4096	2048, 4096, 6144
Downloads	1.9M	12.4M	10.5M	10.5M	80k	0	10K	4.8M	298K	16K

Table 2: Details of evaluated benchmarks. We count benchmarks as containing recursive or indirect calls only if they execute those calls in the critical path of the application. All-Time crates.io download counts are as of 1/5/2023. 'M' and 'A' represent memory and arithmetically intensive benchmarks respectively. 'D' represents benchmarks with substantial divergence. 'Alg' represents benchmarks with significant algorithmic differences. *Bill-PDF uses a no-op system call as a barrier to mitigate heavy program divergence, but does not modify imported libraries.

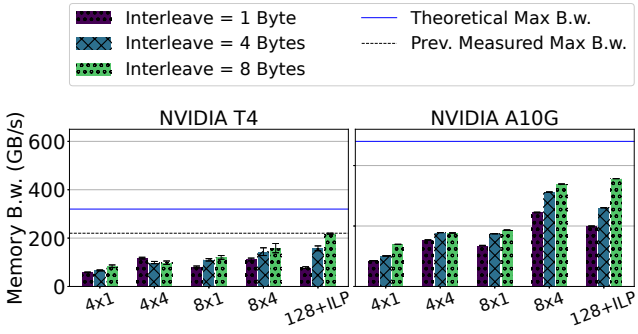


Figure 6: GPU `memcpy` bandwidth (Bytes copied×unroll #)

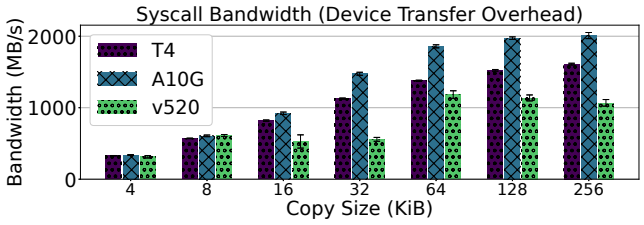


Figure 7: System call host-to-GPU PCIe transfer bandwidth.

4.2 System Performance

4.2.1 Copy Efficiency

Memory Bandwidth. To demonstrate that our memory interleaving can efficiently utilize the high memory bandwidth of GPUs, we evaluate five different `memcpy` implementations which vary copy size (bytes copied per-loop iteration) and loop unroll count. For each configuration we copy 1 MiB of data (using volatile memory accesses to bypass caching effects) from one array in memory to another non-aliased array. Each benchmark is run 50 times, with a heap size of 3 MiB with 4096 (on T4) or 6144 (on A10G) VMs running concurrently. Figure 6 shows that VectorVisor can achieve close to 100% of the experimentally derived maximum memory bandwidth of the T4 [61] and 74% of the theoretical memory bandwidth of the A10G [5]. We can see that larger interleaves, loop unrolling, and instruction level parallelism (ILP) [88] all have substantial impacts on memory bandwidth. VectorVisor leverages the `memory.copy` and `memory.fill` WASM intrinsics to insert optimized copy and fill functions into programs.

Syscall Performance. System calls provide a simple, familiar abstraction for developers to transfer inputs to and from a GPU. However, performing per-VM system calls incurs high data transfer overheads for smaller inputs. To evaluate our system call implementation, we copy inputs to and from the GPU, using batch sizes of 2048 (v520), 4096 (T4), and 6144 (A10G). Figure 7 shows the bandwidth for our VMM excluding network IO. Native CUDA transfer speeds peak at 6.3 GB/s for the T4 and 12.9 GB/s for the A10G—for a single large transfer. Despite high batching overheads, VectorVisor obtains ∼25% of the max possible bandwidth for fine-grained transfers of 256 KiB per-request using the T4. VectorVisor additionally supports overlapping data transfers with running GPU programs to avoid bottlenecks on VMM overhead.

4.2.2 Throughput

We evaluate VectorVisor's throughput against native x86-64 and WebAssembly baselines—with x86-64 being our primary baseline. WebAssembly numbers show the potential for heterogeneous deployments of VectorVisor in addition to showing the overhead of using WebAssembly. Figure 8 shows the best throughput for each configuration that we evaluated in terms of requests per second (RPS). AMD-specific issues, detailed in Section 4.1, prevent some configurations from being evaluated. Detailed throughput results for all system configurations can be found in Table 5 (Appendix).

VectorVisor outperforms x86 and WebAssembly for all but one benchmark (LZ4). We see high variation in throughput for each benchmark across our configurations. Device architecture (e.g., Turing vs. Ampere vs. RDNA 1), memory interleave size, backend compiler optimizations, and program characteristics have outsized impacts on performance. To evaluate the impact of program characteristics such as recursive and indirect function calls, we make use of profile-guided optimization (PGO) to remove indirect calls. Table 3 shows that we can remove all indirect calls from our benchmarks, reducing unoptimized function calls by up to 3430×. Removing all indirect calls and inlining functions where possible obtains mixed results. Bill-PDF improves in throughput by 1.3×, while other benchmarks are 10-20× *slower*. Function inlining caused by removing indirect calls can result in expensive register spilling. Counterintuitively, not removing indirect calls can improve throughput despite the context saving overhead.

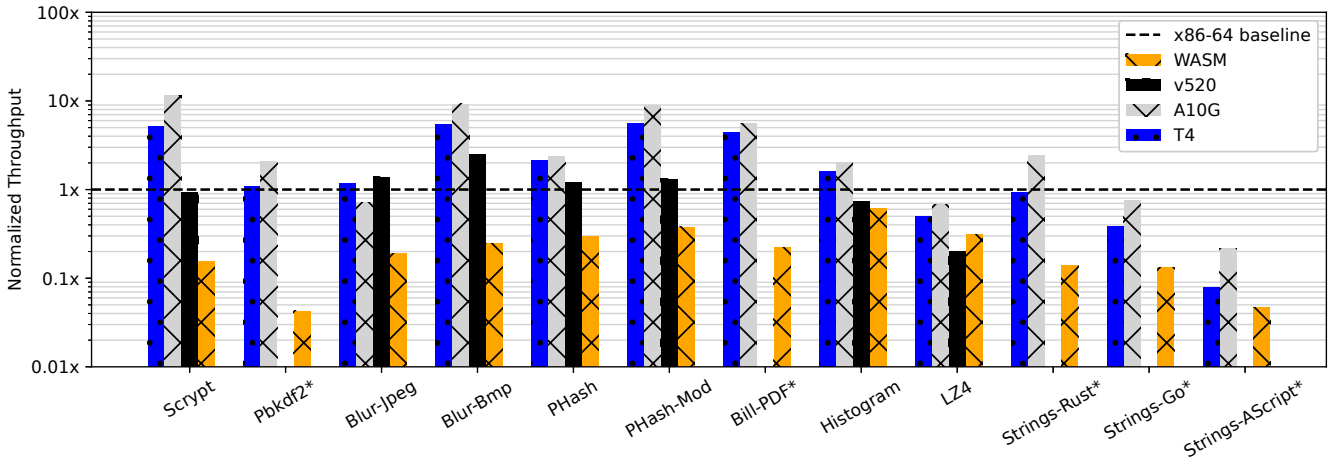


Figure 8: Normalized throughput (average RPS of each benchmark). Results are normalized to the x86-64 baseline for each benchmark except for Strings-Go and Strings-AScript, which are normalized to the x86-64 baseline of Strings-Rust instead. *Benchmarks without an AMD v520 result.

	Scrypt	Pbkdf2	Blur-Jpeg	Blur-Bmp	PHash	PHash-Modified	Bill-PDF	Histogram	LZ4	Strings	Strings-Go	Strings-AssemblyScript
# Total Slowcalls	52062	4828	4023	1416	11460	1410	285632	4117086	804807	43941	2172120	137212574
# Total Slowcalls w/PGO	206	1211	206	213	7844	206	2023	206	206	43744	2990690	143397289
# Indirect Calls	1	5228	162007	2	207031	1	211656	1	1	2	2869093	820142
# Indirect Calls w/PGO	0	0	0	0	0	0	0	0	0	0	0	0

Table 3: *Profile-Guided Optimization Results.* Cumulative indirect and unoptimized call counts for 200 invocations of each instrumented WASM function. These benchmarks were run locally using a 16-core, 64 GB RAM machine running Ubuntu 18.04.

Complex runtimes such as Go and AssemblyScript have significantly higher overhead than Rust on x86-64 WASM baselines (on average 0.41× the throughput of the Rust baseline for Strings-Go using the T4). Runtime support for garbage collection, reflection, and compiler design choices in TinyGo/AssemblyScript all contribute to the observed overheads.

4.2.3 Throughput-per-dollar

Throughput as a metric is insufficient to evaluate VectorVisor. Improving throughput for data parallel workloads by allocating more resources (VMs) represents the status quo. Instead, we show that VectorVisor can achieve greater *efficiency*—improving throughput using fewer resources. Measuring efficiency requires normalizing performance across both CPUs and GPUs, which we accomplish using throughput-per-dollar. It is computed by dividing the requests-per-second (RPS) by the cost of each respective instance per-hour using on-demand pricing in `us-east-1`. On-demand prices are used as a conservative measure of the cost benefits of VectorVisor. Spot instance pricing can be cheaper, further improving the throughput-per-dollar of GPU (T4) instances vs CPU (Intel) instances by 1.49× (reported as of 1/5/2023).

Figure 9 shows the best throughput-per-dollar results for each configuration. Detailed throughput-per-dollar results for all system configurations can be found in Table 6 (Appendix). VectorVisor outperforms x86 instances for four benchmarks (Scrypt, Blur-Bmp, PHash-Modified, Bill-PDF), and on all but two benchmarks versus WebAssembly. Throughput-per-dollar results are overall lower than our throughput results in Section 4.2.2. Leveraging GPU acceleration requires substantial throughput improvements to offset the high cost of GPU hardware (e.g., 3.42× for the T4 vs. AMD x86-64 CPUs). Bottlenecks on application-level divergence (e.g., Strings) and data transfer overheads (e.g., LZ4 and Histogram) result in lower throughput and throughput-per-dollar results.

In three out of the four benchmarks where VectorVisor surpasses our x86-64 baselines, the T4 outperforms the A10G, even though it belongs to an earlier generation of GPUs (e.g., Turing vs. Ampere). Despite differences in GPU hardware, the best predictor of superior throughput-per-dollar with VectorVisor is the ratio of the global memory size (e.g., the number of VMs that can fit) to cost. Compared to the A10G and v520, the T4 packs 27.5% and 43% more VMs-per-dollar respectively. Workloads such as Scrypt, which leverage hardware differences like the larger memory bandwidth of the A10G, can break this trend.

4.2.4 Latency

VectorVisor runs many instances of a program in parallel, improving total throughput, but not latency. Batches of requests have higher on-device execution times than x86-64, ranging from 84-1040× longer using the T4—limiting usage to non-latency sensitive applications.

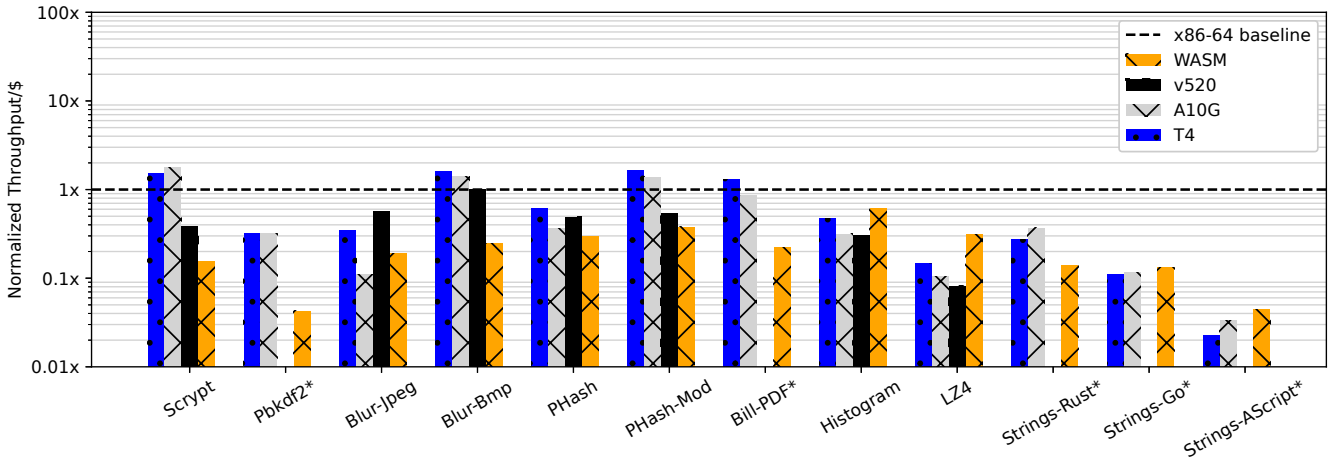


Figure 9: *Benchmark Throughput-per-Dollar.* Results are normalized to the x86-64 baseline for each benchmark except for Strings-Go and Strings-AScript, which are normalized to the x86-64 baseline of Strings-Rust instead. *Benchmarks without an AMD v520 result.

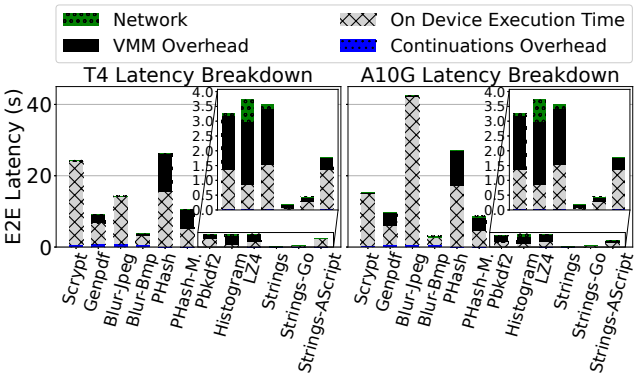


Figure 10: Per-benchmark latency breakdown of execution time, VMM overhead (e.g., syscall overhead), continuations overhead (e.g., context saving/restoring), and network IO. Breakdowns correspond to the best performing configurations with PGO disabled from Table 5.

4.3 Latency breakdown

Figure 10 shows the end-to-end (E2E) latency breakdown for each benchmark. Batch sizes, which impact request latency, can be found in Table 2. On-device execution time dominates the E2E latency for most benchmarks, with the histogram benchmark being the exception. We see that supporting preemption using continuations has low overhead, varying between <1% (PHash-Modified) and 19% (Blur-Bmp) of the on-device execution time. Similarly, by overlapping compute with VMM and network IO VectorVisor significantly reduces related overheads. Benchmarks with a low operational intensity (Ops/Byte) (e.g., Histogram, LZ4) which cannot overlap on-device execution time with batch formation as efficiently are more likely to bottleneck on VMM or network IO.

GPU	Platform	Instance Name	Benchmark	Throughput	Throughput/$
NVIDIA T4	VectorVisor	g4dn.xlarge	Blur-Bmp	804.83	1530.10
NVIDIA A10G	VectorVisor	g5.xlarge	Blur-Bmp	1365.84	1357.69
NVIDIA T4	VectorVisor	g4dn.xlarge	PHash-Modified	384.32	730.65
NVIDIA A10G	VectorVisor	g5.xlarge	PHash-Modified	608.02	604.40
NVIDIA T4	CUDA	g4dn.xlarge	Blur-Bmp	576.28	1095.59
NVIDIA T4	CUDA	g4dn.2xlarge	Blur-Bmp	1118.95	1487.96
NVIDIA A10G	CUDA	g5.xlarge	Blur-Bmp	652.96	649.06
NVIDIA A10G	CUDA	g5.2xlarge	Blur-Bmp	1250.33	1031.62
NVIDIA T4	CUDA	g4dn.xlarge	PHash-Modified	408.85	777.27
NVIDIA T4	CUDA	g4dn.2xlarge	PHash-Modified	821.15	1091.95
NVIDIA A10G	CUDA	g5.xlarge	PHash-Modified	462.27	459.52
NVIDIA A10G	CUDA	g5.2xlarge	PHash-Modified	896.35	739.56

Table 4: Performance of handwritten CUDA benchmarks.

4.3.1 CUDA Comparison

Leveraging GPU acceleration typically involves manually breaking down a program into fine-grained tasks which can be parallelized—speeding up individual invocations of a function. In contrast, VectorVisor runs many instances of the same program in parallel, improving throughput but not latency. To evaluate the efficiency of our translation, we manually rewrote two benchmarks (Blur-Bmp and PHash-Modified) using CUDA. CUDA baselines incur additional CPU overhead from increased kernel launch overheads and running a fraction of the workload on the CPU. To fairly evaluate these baselines, we benchmark them using both xlarge and 2xlarge instances with additional CPUs (Table 1).

We see in Table 4 that VectorVisor slightly outperforms a handwritten CUDA Gaussian blur function, and obtains 67% of the throughput-per-dollar of our CUDA PHash function. PHash-Modified has higher VMM overhead than Blur-Bmp (35% vs 12% of the E2E latency) which affects overall efficiency.

5 Discussion

Workload Characterization. Identifying ideal workloads for VectorVisor is key to improving the cost efficiency of real

applications. Ideal workloads minimize divergent execution, recursion, indirect calls, and are compute-bound. Future work can incorporate model-based approaches [79] to identifying acceleration opportunities for VectorVisor.

Evaluation Limitations. We use both throughput and throughput-per-dollar as evaluation metrics. Throughput-per-dollar is a powerful metric that enables us to compare the end-to-end *efficiency* of VectorVisor, which considers system complexities as well as the capital and operational cost implications of running throughput-oriented workloads. Cloud providers allow customers to insure themselves against high variation in hardware pricing [62, 63], providing a steady baseline cost (at a premium). On-premises hardware configurations can be less expensive over long periods of time, for those willing to pay higher up-front costs. Despite shortcomings, cost-based efficiency metrics provide tangible baselines.

System Call Implementations. Providing support for system calls using continuations was key to running realistic workloads using VectorVisor. Systems such as GPUfs, GPUnet, and Berkeley Borph [76–78, 82, 86] instead provide support using a more performant RPC-like interface using vendor-specific APIs or custom drivers. RPC-style interfaces rely on the ability to perform concurrent and consistent CPU–GPU memory accesses. OpenCL 2.0 *in theory* enables this with fine-grained buffer SVM [3, 53]. In practice, support for fine-grained SVM is mixed—with NVIDIA OpenCL 3.0 not supporting the API and AMD providing partial support [2]. Continuations provide a cross-platform and reasonably performant approach to supporting system call support for GPUs.

6 Related Work

Continuations. Continuations are often used by compilers to support complex control flow operations such as exceptions and preemption [27, 28, 30, 71, 84]. VectorVisor uses continuations to efficiently provide support for preemption and complex control flow on GPUs.

GPU Preemption. GPU kernel preemption can be supported through compiler-based approaches that partition (or slice) programs into chunks [29, 35, 39, 89, 93, 94], or with hardware/driver support [13, 68, 83, 86].

High-Level GPU Languages. CUDA or OpenCL require developers to write programs using low level abstractions. High level language approaches [2, 31, 44, 48, 52, 55, 57, 72, 92] make it easier to accelerate existing programs by reusing existing codebases. Common language features such as dynamic memory allocation, garbage collection, reflection, and recursion are often absent. Unlike VectorVisor, code often must be rewritten to explicitly leverage parallel APIs.

Domain-Specific GPU Systems and Languages. Programming languages designed for domain-specific workloads (DSLs) [24, 38, 40, 46, 58, 73–75, 80] can offer substantially improved performance over general-purpose programming languages. DSLs obtain superior performance through language restrictions, forcing developers to express programs using specific syntax or function calls. While DSLs can efficiently accelerate specific workloads, they trade off performance for programmability—e.g., many workloads cannot be expressed using restrictive DSLs. Similar to DSLs, domain-specific systems can significantly improve performance for throughput-oriented workloads [32, 42, 56, 87]. Domain-specific systems vectorize common workloads (e.g., image processing, machine learning, database operations) using handwritten GPU kernels. Other systems manually vectorize functions from (non GPU-specific) DSLs (i.e. SQL) [50, 51, 69].

Vectorized Program Translation. Systems that abstract a SIMT or SIMD lane as a VM often target restricted use-cases (e.g., fuzz testing) [37, 45, 47, 49]. VectorVisor's design and implementation notably differ from prior work, offering superior GPU language, runtime, and hardware support.

7 Conclusion

VectorVisor is a research prototype which demonstrates that applications originally written for CPUs can be directly run on GPUs without significant modifications. Not only is such GPU execution possible, but it can in fact yield superior throughput-per-dollar versus compute-optimized x86-64 CPUs in the cloud.

Binary translation for GPUs is an exciting and predominantly unexplored area of research, with many potential applications. VectorVisor shows the viability of our new approach to parallelism, opening up the area to future research.

8 Acknowledgements

We would like to thank our shepherd, Redha Gouicem, and the anonymous reviewers for helping us improve this paper. We also thank Rachit Nigam, Mieszko Lis, and Devon Loehr for their valuable comments on earlier versions of it.

References

[1] NVIDIA OpenCL Best Practices Guide. https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, 2009.

[2] Aparapi. https://code.google.com/archive/p/aparapi/, 2011.

[3] OpenCL™ 2.0 Shared Virtual Memory Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/opencl-20-shared-virtual-memory-overview.html, 2014.

[4] AMD OpenCL Programming Optimization Guide. https://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf, 2015.

[2] Fine-grained SVM support is present, but not SVM Atomics.

[5] NVIDIA A10G Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a10/pdf/a10-datasheet.pdf, 2019.

[6] Open Sourcing Photo and Video-Matching Technology to Make the Internet Safer. https://about.fb.com/news/2019/08/open-source-photo-video-matching/, 2019.

[7] Threatexchange. https://github.com/facebook/ThreatExchange, 2019.

[8] AMD EPYC 7002 Processor Datasheet. https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf, 2021.

[9] Blockhash. https://web.archive.org/web/20210827144701/http://blockhash.io/, 2021.

[10] crates.io. https://crates.io/, 2021.

[11] CUDA. https://developer.nvidia.com/cuda-toolkit, 2021.

[12] CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2021.

[13] cuda-gdb. https://docs.nvidia.com/cuda/pdf/cuda-gdb.pdf, 2021.

[14] CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html, 2021.

[15] Scrypt. https://litecoin.info/index.php/Scrypt, 2021.

[16] The OpenCL C Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html, 2021.

[17] wasm-snip. https://github.com/rustwasm/wasm-snip, 2021.

[18] Wasmtime. https://github.com/bytecodealliance/wasmtime, 2021.

[19] AssemblyScript. https://www.assemblyscript.org/, 2022.

[20] Cisco Password Types: Best Practices. https://media.defense.gov/2022/Feb/17/2002940795/-1/-1/1/CSI_CISCO_PASSWORD_TYPES_BEST_PRACTICES_20220217.PDF, 2022.

[21] Radeon Pro v520 ROCm Support. https://github.com/RadeonOpenCompute/ROCm/issues/1706, 2022.

[22] TinyGo. https://github.com/tinygo-org/tinygo, 2022.

[23] ROCm Installation Guide v5.0. https://docs.amd.com/bundle/ROCm_Installation_Guidev5.0/page/Prerequisite_Actions.html, 2023.

[24] 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)*, 2016.

[25] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *European Conference on Object-Oriented Programming (ECOOP)*, 1996.

[26] Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. Tighten Rust's Belt: Shrinking Embedded Rust Binaries. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2022.

[27] Henry G Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *ACM Sigplan Notices*, 1995.

[28] Joel F Bartlett. SCHEME-> C a portable Scheme-to-C compiler. In *WRL Research Report 89/1*, 1989.

[29] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[30] Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. Putting in all the stops: Execution control for JavaScript. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[31] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2018.

[32] Nils Boeschen and Carsten Binnig. GaccO-A GPU-accelerated OLTP DBMS. In *Proc. Intl. Conference on Management of Data (SIGMOD)*, 2022.

[33] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2012.

[34] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, 1994.

[35] Jon Calhoun and Hai Jiang. Preemption of a CUDA Kernel Function. In *13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2012.

[36] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[37] Ahmet Celik, Pengyu Nie, Christopher J Rossbach, and Milos Gligoric. Design, Implementation, and Application of GPU-Based Java Bytecode Interpreters. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2019.

[38] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming (DAMP)*, 2011.

[39] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[40] 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)*, 2018.

[41] Zhiyuan Cheng, James Caverlee, and Kyumin Lee. You are where you tweet: a content-based approach to geo-locating Twitter users. In *19th ACM International Conference on Information and Knowledge Management (CIKM)*, 2010.

[42] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[43] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. Divergence analysis and optimizations. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[44] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F Bacon, and Stephen J Fink. Compiling a High-Level Language for GPUs: (via language support for architectures and compilers). In *33th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[45] Ryan Eberhardt, Artem Dinaburg, and Peter Goodman. Let's build a high-performance fuzzer with GPUs! https://blog.trailofbits.com/2020/10/22/lets-build-a-high-performance-fuzzer-with-gpus/, 2020.

[46] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: DSL for linear algebra and neural net computations on GPUs. In *2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2018.

[47] Brandon Falk. Vectorized Emulation: Hardware accelerated taint tracking at 2 trillion instructions per second. https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html, 2018.

[48] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-in-time GPU compilation for interpreted languages with partial evaluation. In *13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2017.

[49] Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. Running parallel bytecode interpreters on heterogeneous hardware. In *4th International Conference on Art, Science, and Engineering of Programming (Programming)*, 2020.

[50] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In *Proc. Intl. Conference on Management of Data (SIGMOD)*, 2018.

[51] Henning Funke and Jens Teubner. Data-parallel query processing on non-uniform data. *Proceedings of the VLDB Endowment*, 2020.

[52] Kate Gregory and Ade Miller. *C++ AMP: accelerated massive parallelism with Microsoft Visual C++*. Microsoft Press, 2012.

[53] Khronos OpenCL Working Group. The OpenCL Specification Version: 2.1 Document Revision: 24. https://registry.khronos.org/OpenCL/specs/opencl-2.1.pdf#page=174, 2018.

[54] 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. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[55] Michael Haidl and Sergei Gorlatch. PACXX: Towards a unified programming model for programming accelerators using C++ 14. In *LLVM Compiler Infrastructure in HPC*, 2014.

[56] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. MemcachedGPU: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[57] Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine, and Nicholas D Matsakis. GPU programming in Rust: Implementing high-level abstractions in a systems-level language. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*, 2013.

[58] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable stream programming on graphics engines. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011.

[59] 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 ACM on Programming Languages (OOPSLA)*, 2000.

[60] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. Accelerating SSL with GPUs. *ACM SIGCOMM Computer Communication Review*, 2010.

[61] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVidia Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[62] Michael Kan. Read it and weep: Here's how bad Nvidia GPU prices got in a single year. https://www.pcmag.com/news/read-it-and-weep-heres-how-bad-nvidia-gpu-prices-got-in-a-single-year, 2021.

[63] Michael Kan. Corsair: Expect some GPU prices to dip 'below msrp' soon as prices normalize. https://www.pcmag.com/news/corsair-expect-some-gpu-prices-to-dip-below-msrp-soon-as-prices-normalize, 2022.

[64] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *In International Symposium on Code Generation and Optimization (CGO)*, 2004.

[65] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2008.

[66] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H Loh, Mahesh Subramony, and Sean White. Pioneering chiplet technology and design for the AMD EPYC™ and Ryzen™ processor families : Industrial product. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[67] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Pipelined parallel LZSS for streaming data compression on GPGPUs. In *IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[68] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2015.

[69] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proceedings of the VLDB Endowment*, 2020.

[70] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle inventor: data movement synthesis for gpu kernels. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[71] Donald Pinckney, Arjun Guha, and Yuriy Brun. Wasm/k: delimited continuations for WebAssembly. In *16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS)*, 2020.

[72] Philip C Pratt-Szeliga, James W Fawcett, and Roy D Welch. Rootbeer: Seamlessly using GPUs from Java. In *IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012.

[73] 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)*, 2013.

[74] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *23th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2011.

[75] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2013.

[76] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[77] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. GPUnet: Networking abstractions for GPU programs. *ACM Transactions on Computer Systems (TOCS)*, 2016.

[78] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

[79] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[80] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013.

[81] Gerald Jay Sussman and Guy L Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 1998.

[82] Yusuke Suzuki, Hiroshi Yamada, Shinpei Kato, and Kenji Kono. GLoop: an event-driven runtime for consolidating GPGPU applications. In *8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[83] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2014.

[84] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1992.

[85] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in Rust. *Proceedings of the ICSE-SEIP*, 2022.

[86] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. Generic system calls for GPUs. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[87] Matthias Vogelgesang, Suren Chilingaryan, Tomy dos Santos Rolo, and Andreas Kopmann. UFO: A scalable GPU-based image processing framework for on-line monitoring. In *IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems*, 2012.

[88] Vasily Volkov. Better performance at lower occupancy. In *GPU Technology Conference, GTC*, 2010.

[89] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. FLEP: Enabling flexible and efficient preemption on GPUs. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017.

[90] Bian Yang, Fan Gu, and Xiamu Niu. Block mean value based image perceptual hashing. In *IEEE International Conference on Intelligent Information Hiding and Multimedia*, 2006.

[91] Alon Zakai. Binaryen. https://github.com/WebAssembly/binaryen, 2022.

[92] Wojciech Zaremba, Yuan Lin, and Vinod Grover. JaBEE: framework for object-oriented Java bytecode compilation and execution on graphics processor units. In *5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, 2012.

[93] Lior Zeno, Avi Mendelson, and Mark Silberstein. GPUpIO: The case for I/O-driven preemption on GPUs. In *9th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU)*, 2016.

[94] Husheng Zhou, Guangmo Tong, and Cong Liu. GPES: A preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.

A Appendix

A.1 Tables

System	Platform	PGO	Interleave	Scrypt	Pbkdf2	Blur Jpeg	Blur Bmp	PHash	PHash Mod.	Bill PDF	Histogram	LZ4	Strings (Rust / Go / AScript)
VectorVisor	AMD v520	Y	4	32.27	N/A	202.24	368.64	70.28	77.85	N/A	832.43	449.89	N/A / N/A / N/A
VectorVisor	AMD v520	Y	8	29.29	N/A	**245.76**	N/A	79.67	89.86	N/A	848.31	N/A	N/A / N/A / N/A
VectorVisor	NVIDIA T4	N	4	109.32	71.09	209.94	726.59	129.18	341.73	339.06	1570.90	818.90	9535.47 / 4112.16 / 861.24
VectorVisor	NVIDIA T4	N	8	160.25	1355.05	177.07	804.83	140.44	367.31	380.55	1730.05	1114.30	10242.08 / 3735.93 / 826.29
VectorVisor	NVIDIA T4	Y	4	108.51	63.68	50.66	720.79	10.44	349.98	398.26	1827.91	371.75	9077.32 / 4204.24 / 845.07
VectorVisor	NVIDIA T4	Y	8	179.54	59.49	60.59	728.86	10.90	384.32	497.42	1676.06	486.94	10332.82 / 3842.04 / 841.04
VectorVisor	NVIDIA A10G	N	4	297.29	**2596.52**	93.30	1237.87	153.99	546.35	480.29	886.25	1527.21	24543.24 / **8438.01** / 1781.76
VectorVisor	NVIDIA A10G	N	8	389.08	168.56	69.52	**1365.84**	**158.99**	592.87	509.71	484.22	1490.67	25981.84 / 8030.55 / 1965.93
VectorVisor	NVIDIA A10G	Y	4	297.08	145.66	71.73	1166.44	14.81	533.85	308.62	**2356.72**	722.01	24109.98 / 8377.06 / **2398.01**
VectorVisor	NVIDIA A10G	Y	8	**397.10**	143.55	127.47	1163.57	20.87	**608.02**	**619.04**	517.97	945.06	**26598.05** / 8302.31 / 1977.61
CPU (x86-64)	AMD	N/A	N/A	33.89	1233.44	176.63	147.14	66.05	68.02	111.18	1140.20	**2235.93**	11002.53 / N/A / N/A
CPU (x86-64)	Intel	N/A	N/A	34.27	149.33	148.53	135.43	55.83	58.00	85.95	1144.96	1987.63	10182.98 / N/A / N/A
CPU (WASM)	AMD	N/A	N/A	5.33	52.67	33.81	36.13	19.86	25.78	24.97	697.94	700.77	1536.22 / 1450.18 / 485.62
CPU (WASM)	Intel	N/A	N/A	4.35	46.49	25.83	17.33	12.13	13.70	21.44	659.40	596.05	1431.18 / 1375.47 / 514.84

Table 5: Average requests per second (RPS) of each benchmark. Bold values correspond to the best throughput.

System	Platform	PGO	Interleave	Scrypt	Pbkdf2	Blur Jpeg	Blur Bmp	PHash	PHash Mod.	Bill PDF	Histogram	LZ4	Strings (Rust / Go / AScript)
VectorVisor	AMD v520	Y	4	85.27	N/A	534.32	973.95	185.68	205.69	N/A	2199.28	1188.61	N/A / N/A / N/A
VectorVisor	AMD v520	Y	8	77.38	N/A	649.30	N/A	210.49	237.41	N/A	2241.24	N/A	N/A / N/A / N/A
VectorVisor	NVIDIA T4	N	4	207.84	135.16	399.13	1381.34	245.58	649.67	644.60	2986.51	1556.85	18128.26 / 7817.80 / 1637.33
VectorVisor	NVIDIA T4	N	8	304.67	2576.15	336.63	**1530.10**	266.99	698.31	723.48	3289.07	2118.43	19471.63 / 7102.54 / 1570.90
VectorVisor	NVIDIA T4	Y	4	206.30	121.06	96.31	1370.32	19.85	665.35	757.15	3475.12	706.74	17257.26 / 7992.86 / 1606.60
VectorVisor	NVIDIA T4	Y	8	341.32	113.11	115.20	1385.66	20.73	**730.65**	**945.67**	3186.43	925.75	19644.15 / 7304.26 / 1598.94
VectorVisor	NVIDIA A10G	N	4	295.52	2581.03	92.74	1230.48	153.07	543.09	477.42	880.96	1518.10	24396.86 / 8387.68 / 1771.13
VectorVisor	NVIDIA A10G	N	8	386.76	167.56	69.11	1357.69	158.04	589.33	506.67	481.33	1481.78	25826.88 / 7982.65 / 1954.21
VectorVisor	NVIDIA A10G	Y	4	295.31	144.79	71.30	1159.48	14.72	530.66	306.78	2342.66	717.70	23966.18 / 8327.10 / 2383.71
VectorVisor	NVIDIA A10G	Y	8	**394.73**	142.69	126.71	1156.63	20.74	604.40	615.35	514.88	939.43	26439.41 / 8252.79 / 1965.81
CPU (x86-64)	AMD	N/A	N/A	220.08	**8009.37**	**1146.95**	955.45	**428.87**	441.67	721.94	**7403.91**	**14519.04**	**71444.98** / N/A / N/A
CPU (x86-64)	Intel	N/A	N/A	201.60	878.40	873.70	796.63	328.39	341.18	505.61	6735.05	11691.97	59899.89 / N/A / N/A
CPU (WASM)	AMD	N/A	N/A	34.60	342.03	219.52	234.62	128.97	167.39	162.16	4532.09	4550.45	9975.48 / **9416.76** / **3153.38**
CPU (WASM)	Intel	N/A	N/A	25.58	273.47	151.95	101.95	71.34	80.58	126.13	3878.82	3506.18	8418.72 / 8091.03 / 3028.45

Table 6: *Benchmark Throughput-per-Dollar.* Values correspond to the average RPS of each benchmark normalized to instance cost. Bold values correspond to the best throughput-per-dollar.

A.2 Rust Example

```rust
#[macro_use]
extern crate lazy_static;
// Import existing open-source libraries!
use pdf_writer::*;
use pdf_writer::types::{ActionType, AnnotationType, BorderType};
use std::fs::File;
use std::io::Write;
use std::time::Instant;
// Import our custom 'serverless' runtime. We use this in our x86 and WASM benchmarks as well.
// WASM benchmarks run the same binary that VectorVisor does!
use wasm_serverless_invoke::wasm_handler::*;
use wasm_serverless_invoke::wasm_handler::WasmHandler;
use wasm_serverless_invoke::wasm_handler::SerializationFormat::MsgPack;
use serde::Deserialize;
use serde::Serialize;
// Image and compression libraries
use image::{ColorType, GenericImageView, ImageFormat};
use miniz_oxide::deflate::{compress_to_vec_zlib, CompressionLevel};
// Include a sample template image for our PDF footer
lazy_static! {
    static ref EMBED_IMAGE: &'static [u8] = include_bytes!("test.png");
}
// Syntactic sugar for (de)serializing JSON/MsgPack inputs
#[derive(Debug, Deserialize)]
struct FuncInput {
    name: String,
    purchases: Vec<String>,
    price: Vec<f64>, // Typically prices should not be encoded as floats, we do this for simplicity.
}
#[derive(Debug, Deserialize)]
struct BatchInput {
    inputs: Vec<FuncInput>
}
#[derive(Debug, Serialize)]
struct FuncResponse {
    resp: Vec<u8>
}
#[derive(Debug, Serialize)]
struct BatchFuncResponse {
    resp: Vec<FuncResponse>
}
#[inline(never)]
fn makePdf(event: FuncInput) -> Vec<u8> {
    // Perform PDF formatting, image manipulation, and compression to generate a valid PDF
}
fn batch_genpdf(inputs: BatchInput) -> BatchFuncResponse {
    let mut results = vec![];
    for input in inputs.inputs {
        results.push(FuncResponse { resp: makePdf(input) });
        // Bill-PDF is the only benchmark to use system calls as synchronization barriers
        unsafe { vectorvisor_barrier() }; // We can wait on arbitrary subsets of VMs (unlike OpenCL barrier(...))
    }
    return BatchFuncResponse{ resp: results };
}
fn main() {
    // Specify input format type and buffer sizes
    let handler = WasmHandler::new(&batch_genpdf);
    // Starts the event-loop and encapsulates serverless_invoke/serverless_response
    handler.run_with_format(1024*512, MsgPack);
}
```

Figure 11: *Bill-PDF*. This benchmark performs PDF processing, image manipulation, and compression.

A.3 Golang Example

```go
package main;

// define our system call interface
// #include "serverless.c"
import "C"
import (
    // Import JSON + string manipulation libraries
    "github.com/json-iterator/tinygo"
    "unsafe"
    "strings"
)

//go:generate go run github.com/json-iterator/tinygo/gen
type Payload struct {
    Tweets  []string `json:"tweets"`
}
//go:generate go run github.com/json-iterator/tinygo/gen
type Response struct {
    Tokenized  [][]string
    Hashtags   [][]string
}
// Go doesn't provide Map/Filter for us, so we use our own implementation
func Map[T, U any](ts []T, f func(T) U) []U {
    ...
}
func Filter(vs []string, f func(string) bool) []string {
    ...
}
func main() {
    json := jsoniter.CreateJsonAdapter(Payload_json{}, Response_json{})
    // Use this as a set, track all stopwords
    stopwordSet := make(map[string]bool)
    for _, word := range stopWords {
        stopwordSet[word] = true
    }
    input_buf := make([]byte, 1024*450) // buffer for raw inputs from VectorVisor
    for { // serverless_invoke is the system call used for transferring inputs from the host (CPU) to the GPU
        in_size := C.serverless_invoke((*C.char)(unsafe.Pointer(&input_buf[0])), 1024*450)
        if in_size == 0 { // if in_size == 0, then this VM is blocked off and has no input for this batch
            fakeaddr := uintptr(0x0) // serverless_response copies inputs from the GPU back to the CPU.
            C.serverless_response((*C.char)(unsafe.Pointer(fakeaddr)), 0)
            continue
        }
        var input Payload;
        json.Unmarshal(input_buf[0:in_size], &input);
        // First tokenize each tweet []string --> [][]string
        ...
        // Now process each tweet, filtering out stop words
        ...
        // Get the hashtags, we will add them as we see them
        var tags = make([][]string, 0)
        ...
        var response Response;  // create a JSON response and return it!
        response.Tokenized = tokenized;
        response.Hashtags = tags;
        bytes, _ := json.Marshal(response);
        C.serverless_response((*C.char)(unsafe.Pointer(&bytes[0])), (C.uint)(len(bytes)))
    }
}
```

Figure 12: *Strings-Go*. Tokenize some input tweets and return the hashtags. TinyGo (https://tinygo.org/docs/reference/lang-support/) provides us with a conservative mark and sweep garbage collector, limited runtime reflection and goroutine support.

A.4 AssemblyScript Example

```
import { Console, FileSystem, Descriptor } from "as-wasi/assembly"; // Import needed syscalls
import { JSON, JSONEncoder } from "assemblyscript-json/assembly"; // Import a JSON encoder/decoder
import { listen } from "./env"; // Import our event-driven runtime
import { stopWords, initSet, getSet } from "./stop"; // Import a dataset of stopwords
function abort(message: usize, fileName: usize, line: u32, column: u32): void {
    unreachable() // needed for the AssemblyScript runtime
}
initSet(); // init our set of "stop words"
let set: Set<string> = getSet();

// TypeScript-like syntax for GPU programing!
function process_tweets(input: JSON.Obj): Uint8Array | null {
    let tweets: JSON.Arr | null = input.getArr("tweets");
    if (tweets != null) {
        let strTweets: string[] = tweets._arr.map<string>((val: JSON.Value): string => val.toString());
        // Split each tweet (tokenize)
        let tokenize: string[][] = strTweets.map<string[]>((val: string): string[] => val.split("␣"));
        // Remove empty values and stop words
        let filtered: string[][] = tokenize.map<string[]>((arr: string[]): string[] =>
                                    arr.filter((word: string): bool => {
                                            if (set.has(word)) {
                                                return false;
                                            } else {
                                                return true;
                                            }
                                        }));
        // Get the array of hashtags for each tweet
        let hashtags: string[][] = filtered.map<string[]>((tweet: string[]): string[] =>
                                    tweet.filter((word: string): bool => {
                                        if (word.charAt(0) == '#' && word.charAt(1) != "") {
                                            return true;
                                        } else {
                                            return false;
                                        }
                                    }));
        let encoder = new JSONEncoder(); // encode a JSON response
        encoder.pushArray("tokenized");
        for (let tweet_idx = 0; tweet_idx < filtered.length; tweet_idx++) {
            ...
        }
        encoder.popArray();
        encoder.pushArray("hashtags");
        for (let tweet_idx = 0; tweet_idx < hashtags.length; tweet_idx++) {
            ...
        }
        encoder.popArray();
        let json: Uint8Array = encoder.serialize();
        return json;
    }
    // else we failed somehow...
    return null;
}
listen(1024*512, process_tweets); // Starts the event-loop and encapsulates serverless_invoke/serverless_response
```

Figure 13: *Strings-AssemblyScript*. Same as Strings+Strings-Go, but with different syntax. Support for incremental garbage collection is provided.