UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Advanced Software Techniques for Emerging Memory Technologies

Permalink

https://escholarship.org/uc/item/6g48430x

Author

Kim, Juno

Publication Date

2023

Peer reviewed|Thesis/dissertation

eScholarship.org

Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Advanced Software Techniques for Emerging Memory Technologies

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Juno Kim

Committee in charge:

Professor Steven Swanson, Chair
Professor Paul H. Siegel
Professor Geoffrey Voelker
Professor Jishen Zhao

2023

Copyright

Juno Kim, 2023

All rights reserved.

The Dissertation of Juno Kim is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

To my wife Jun Hee, who has been there with me since the beginning of this journey. And to my beloved daughter Bomi whose birth was the most fabulous graduation present from God and the most influential encouragement to me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to express my deep and sincere gratitude to my research advisor, Dr. Steven Swanson, for giving me the chance to work with him and supporting me throughout my Ph.D. study. I am especially thankful for his patience and kind support under various circumstances. His mentorship has helped me in all the time of research to grow academically. I could not have imagined having a better advisor and mentor for my Ph.D. study.

I want to express my appreciation to my committee members, Geoffrey Voelker, Jishen Zhao, and Paul Siegel for their constructive guidance and feedback. Your suggestions have been an invaluable resource throughout this process.

I want to thank my colleagues at the Non-Volatile Systems Laboratory (NVSL). This dissertation is an outcome of hard works with many past and present NVSL members. I am also grateful to Joseph Izraelevitz and Terence Kelly who offered enough time and resources to help my research throughout discussions.

Finally, I want to express my great appreciation to my family and friends. Specifically, my successful end of this journey is deeply indebted to my wife Jun Hee who has always been by my side regardless of ups and downs. I am forever indebted to my parents and brother for their unconditional love and support.

Chapter 1, Chapter 2, and Chapter 3 contain material from "Finding and Fixing Performance Patholo- gies in Persistent Memory Software Stacks", by Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson, which appeared in the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019). The dissertation author is the primary investigator and the co-first author of this paper.

Chapter 1, Chapter 2, and Chapter 4 contain material from "SubZero: Zero-Copy IO for Persistent Main Memory File Systems", by Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson, which appeared in the 11th ACM SIGOPS

Asia-Pacific Workshop on Systems (APSys 2020). The dissertation author is the primary investigator and the first author of this paper.

Chapter 1, Chapter 2, and Chapter 5 contain material from “Blaze: Fast Graph Processing on Fast SSDs”, by Juno Kim and Steven Swanson, which appeared in the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2022). The dissertation author is the primary investigator and the first author of this paper.

Chapter 1, Chapter 2, and Chapter 6 contain material from “TOSS: Tiering of Serverless Snapshots for Low Cost Serverless Computing”, by Juno Kim, Theodoros Michailidis, Linsong Guo, Jishen Zhao, and Steven Swanson, which is in preparation for submission to the European Conference on Computer Systems (EuroSys 2024). The dissertation author is the primary investigator and the co-first author of this paper.

Permission to make digital or hard copies of part of all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage

VITA

2012	Bachelor of Science in in Electrical and Computer Engineering, University of California San Diego
2012–2014	Software Engineer, SAP Labs Korea
2017–2020	Master of Science in Computer Science, University of California San Diego
2019	Internship, IBM Research
2021	Internship, Intel
2022	Internship, Intel Labs
2017-2023	Doctor of Philosophy in Computer Science, University of California San Diego

PUBLICATIONS

Juno Kim and Steven Swanson, "Blaze: Fast Graph Processing on Fast SSDs", In the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2022.

Hanxian Huang, Zixuan Wang, Juno Kim, Steven Swanson and Jishen Zhao, "Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming", In the USENIX Annual Technical Conference (ATC), 2021.

Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao and Steven Swanson, "Sub Zero: Zero copy IO for Persistent Main Memory File Systems", In the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys), 2020 (Awarded Best Paper).

Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz and Steven Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory", In the 18th USENIX Conference on File and Storage Technologies (FAST), 2020.

Jian Xu, Juno Kim, Amirsaman Memaripour and Steven Swanson, "Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks", In the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

Joshua Lockerman, Jose Faleiro, Juno Kim, Soham Sankaran, Daniel Abadi, James Aspnes, Siddhartha Sen and Mahesh Balakrishnan, "The FuzzyLog: A Partially Ordered Shared Log", In the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018.

ABSTRACT OF THE DISSERTATION

Advanced Software Techniques for Emerging Memory Technologies

by

Juno Kim

Doctor of Philosophy in Computer Science

University of California San Diego, 2023

Professor Steven Swanson, Chair

Emerging memory technologies, such as persistent memory and ultra-low-latency SSDs, change the tradeoffs system designers must consider. As storage, they offer considerably faster persistence than traditional SSDs and hard disks. As memory, they expand the capacity of main memory in a single machine at a lower price than conventional, expensive DRAM. Despite these advantages, efficiently leveraging them remains challenging in a variety of software stacks including file systems, databases, graph processing systems, and serverless computing.

This dissertation first presents a comprehensive study that reveals the impact of state-of-the-art persistent memory (PMem) storage systems on IO-intensive applications and explores how those applications can best leverage the performance of PMem. It examines various performance problems that PMem introduces at both the file system and application level, and suggests potential optimization paths while discussing their different tradeoffs.

Second, the dissertation introduces SubZero, a novel file system API that allows fast and easy access to files residing in PMem file systems. SubZero offers `mmap()`-like performance for both read and write while providing similar isolation semantics as `read()` and `write()`, taking the best of both worlds from file-oriented and memory-oriented IO for efficient access to PMem-resident files.

Third, it presents Blaze, a new out-of-core graph processing framework optimized for ultra-low-latency SSDs. While current out-of-core systems cannot fully leverage the high IOPs these devices can deliver due to the high CPU overhead, Blaze achieves high performance on these SSDs with its novel, lightweight parallel value propagation technique using a highly concurrent bin data structure, overcoming the limitations of traditional parallel graph computation techniques such as synchronization or message passing.

Finally, it introduces TOSS, our ongoing effort to support a cost-efficient serverless offering based on memory tiering. TOSS aims to maximize the use of slow memory (e.g., PMem, CXL memory) for the execution of serverless functions while achieving similar performance to the executions solely backed by fast memory (e.g., DRAM). Our evaluation shows that TOSS can offload a significant portion of guest VM memory pages to slow memory while achieving minimal slowdown on a variety of serverless workloads.

Chapter 1

Introduction

Emerging memory technologies, such as persistent memory and ultra-low-latency SSDs, change the tradeoffs system designers must consider. As storage, they offer considerably faster persistence than traditional SSDs and hard disks. As memory, they expand the capacity of main memory in a single machine at a lower price than conventional, expensive DRAM. However, software designed with conventional design principles in mind is likely to be a poor fit for these emerging memory technologies.

From the perspective of system designers, either memory technologies offer the following key benefits. First, they offer fast persistence from user-space, making unnecessary to access persistent storage from the kernel. Second, they offer fine-grained, byte-addressable access to persistent storage unlike traditional block storage. Third, they offer the capability of tiered memory by using them along with traditional fast DRAM. Fourth, they offer considerably increased bandwidth with higher parallelism than traditional block devices. Despite these advantages, efficiently leveraging them remains challenging in a variety of software stacks including file systems, databases, graph processing systems, and serverless computing.

This dissertation focuses on providing software techniques that best leverage emerging memory technologies in the aforementioned software stacks. These techniques enable data-intensive applications and system software to fully utilize the performance, capacity, direct-accessibility of emerging memories with minimized burden to programmers. The techniques it introduces include general solutions implemented in system software layer such as file system as well as domain-specific solutions aimed for user applications such as databases or graph processing systems. In addition, the dissertation describes how emerging memories can be used to lower the cost of datacenter applications by supporting memory tiering in the serverless computing stack.

In Chapter 3, we introduce a comprehensive study that reveals the impact of state-of-the-art persistent memory (PMem) storage systems on IO-intensive applications and explores how those applications can best leverage the performance of PMem. It shows that simple techniques that move file operations into user space dramatically improve application performance. Also, it examines the scalability of PMem file systems in light of the rising core counts and pronounced NUMA effects in modern systems.

In Chapter 4, we present SubZero, a novel file system API that allows fast and easy access to PMem-resident files. Unlike traditional file-oriented IO such as `read()` and `write()` that semantically require a copy of data as buffer, SubZero allows access to PMem files without any copies. Unlike memory-oriented IO such as `mmap()` that requires developers to implement custom isolation mechanisms, SubZero ensures that any concurrent accesses to the same file do not leave the file contents in an inconsistent state. In short, SubZero takes the best of both worlds from file-oriented and memory-oriented IO to offer fast and easy access to the files residing in PMem file systems.

In Chapter 5, we present Blaze, a new out-of-core graph processing framework optimized for ultra-low-latency SSDs. Current out-of-core systems cannot fully leverage the high IOPs these devices can deliver as they often become CPU-bound. The core problem is that traditional value propagation techniques used in current systems – either shared-memory synchronization or message passing – incur high CPU overhead or load imbalance. Blaze overcomes this problem with its novel value propagation technique called Online Binning that leverages a concurrent bin data structure for fast and efficient value propagation among graph vertices, achieving significant speedups over current systems when running various graph workloads on ultra-low-latency SSDs.

In Chapter 6, we introduce TOSS, our ongoing effort to support a cost-efficient serverless offering based on memory tiering. TOSS aims to maximize the use of slow memory (e.g., PMem, CXL memory) for the execution of serverless functions while achieving similar performance to the executions solely backed by fast memory (e.g., DRAM). Our evaluation shows that TOSS can offload a significant portion of guest VM memory pages to slow memory while achieving minimal slowdown on a variety of serverless workloads.

Finally, we conclude this dissertation in Chapter 7 by summarizing its contributions including a comprehensive study of performance problems and optimizations for persistent memory software stacks, a new file system API for persistent memory file systems, a novel graph processing framework optimized for ultra-low-latency SSDs, and a new serverless offering backed by tiered memories for cost-efficient executions of serverless functions.

Acknowledgements

This chapter contains material from “Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks”, by Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson, which appeared in the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019). The dissertation author is the primary investigator and the co-first author of this paper.

This chapter contains material from “SubZero: Zero-Copy IO for Persistent Main Memory File Systems”, by Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson, which appeared in the 11th ACM SIGOPS Asia-Pacific Workshop on

Systems (APSys 2020). The dissertation author is the primary investigator and the first author of this paper.

This chapter contains material from “Blaze: Fast Graph Processing on Fast SSDs”, by Juno Kim and Steven Swanson, which appeared in the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2022). The dissertation author is the primary investigator and the first author of this paper.

This chapter contains material from “TOSS: Tiering of Serverless Snapshots for Low Cost Serverless Computing”, by Juno Kim, Theodoros Michailidis, Linsong Guo, Jishen Zhao, and Steven Swanson, which is in preparation for submission to the European Conference on Computer Systems (EuroSys 2024). The dissertation author is the primary investigator and the co-first author of this paper.

Chapter 2

Background and Motivation

Emerging memory technologies, such as persistent memory and ultra-low-latency SSDs, change the tradeoffs system designers must consider. As storage, they offer considerably faster persistence than traditional SSDs and hard disks. As memory, they expand the capacity of main memory in a single machine at a lower price than conventional, expensive DRAM.

The rest of this chapter provides the background of this dissertation. Section 2.1 offers a brief summary of persistent memory technologies. Section 2.2 reviews the evolution of SSDs with a brief performance comparison. Section 2.3 describes the challenges in leveraging these memory technologies from both persistent storage and volatile memory perspectives.

2.1 Persistent Memory

Persistent memory is a new class of memory attached to attached to the processor memory bus that appear as a directly-addressable, persistent region in the processor's address space. Modern server platforms have supported persistent memory in the form of battery-backed NVDIMMs [75, 47] for several years, and Linux and Windows include facilities to access these memories and build file systems on top of them.

Denser persistent memories that do not need a battery have been announced

Table 2.1. The evolution of storage bandwidth.

SSD	Model	Seq. 4 kB read	Rand. 4 kB read
NAND	Intel SSD DC S3520 (2016)	386 MB/s	132 MB/s
Optane	Intel Optane SSD DC P4800X (2017)	2550 MB/s	2360 MB/s
Z-NAND	Samsung 983 ZET (2018)	3400 MB/s	3072 MB/s
V-NAND	Samsung 980 Pro (2020)	3500 MB/s	2827 MB/s

by Intel and Micron and use a technology termed "3D XPoint" [74]. There are several potential competitors to 3D XPoint, such as spin-torque transfer RAM (STT-RAM) [58, 79], phase change memory (PCM) [22, 28, 65, 86], and resistive RAM (ReRAM) [39, 100]. Each has different strengths and weaknesses: STT-RAM can meet or surpass DRAM's latency and it may eventually appear in on-chip, last-level caches [120], but its large cell size limits capacity and its feasibility as a DRAM replacement. PCM, ReRAM, and 3D XPoint are denser than DRAM, and may enable very large, non-volatile main memories. Their latency will be worse than DRAM, however, especially for writes. All of these memories suffer from potential wear-out after repeated writes.

2.2 Evolution of SSDs

Recent advances in storage technology present new challenges and opportunities for the design of modern data processing systems. The most notable performance trend in storage device is *symmetric high bandwidth*. For instance, Intel Optane SSD [3] achieves about 2.5 GB/s of read bandwidth for both sequential and random 4 kB access while Samsung's Z-NAND [6] and V-NAND SSD [5] show similar performance characteristics (Table 2.1). On conventional NAND SSD, however, random 4 kB read performs only 34% of sequential read bandwidth, showing a large performance gap between random and sequential access. In addition, the absolute bandwidth of modern SSDs has undergone a significant improvement – Optane SSD shows 6.6$\times$ and 17.9$\times$ higher bandwidth than NAND SSD in sequential and random read, respectively. In summary, the huge improvement in storage performance opens new challenges and opportunities in building performant data processing systems.

2.3 Challenges of Using Emerging Memories

The problem with these new memory/storage technologies is that it is challenging to best leverage them in modern data processing systems because they introduce new assumptions, which potentially requires a complete redesign of existing software stacks.

Then what are the new assumptions that emerging memories introduce? First, storage is now byte-addressable thanks to the persistent memory technology. Before persistent memory, storage has always been block-based. Second, storage can be accessed directly from user-space without going through the kernel. Third, storage performance is now quite close to memory. For instance, the performance gap between DRAM and PMem, a commercially available persistent memory product from Intel, is no longer orders of magnitude, and their performance numbers are in the same order. Finally, different memory devices with different performance/price tradeoffs can constitute a multi-tier memory hierarchy, opening new opportunities to lower the cost of memory from a single tier, expensive DRAM.

2.3.1 As Persistent Storage

Based on the new assumtions, the most interesting and innovative challenge is to use persistent memory as persistent storage. Along this path, the first question is how system software such as file systems, or more broadly operating systems, must evolve to accommodate the capabilities of persistent memory. There have been significant efforts to build PMem-aware file systems, either by adapting conventional, disk-based file systems such as Ext4 or XFS, or by designing new file systems tailored for PMem, but the question remains as to which approach is a better solution in terms of performance, reliability, and maintainability going forward.

Another question is how storage applications can be further optimized for persistent memory to best serve their demands. This is a unique challenge in that persistent memory allows applications to access it directly from user space using load/store instructions (so-called DAX capability) unlike conventional storage devices that always require the intervention of file systems to access persistent data. With DAX, application developers can design custom, crash-consistent PMem data structures that fit their needs.

2.3.2 As Volatile Memory

Using emerging memories as volatile memory helps expand the capacity of memory while lowering the cost of it. This dissertation explores the capacity expansion aspect of emering memories in two directions. The first direction is to use fast SSDs to expand the memory capacity. The key challenge in this direction is to optimize IO for accessing data residing in SSDs. The second direction is to slow tier of memories (e.g., PMem, CXL memory) to expand the memory capacity. Unlike using SSD as memory, using slow memories allows more seamless integration with fast memory as most memory vendors provide a rich set of software ecosystems (e.g., OS drivers, userspace libraries) to ease the use of slow memory in existing software stacks. Nevertheless, the key challenge is to identify the memory access pattern of given workloads and optimize the placement of pages in the right tier of memory.

Acknowledgements

This chapter contains material from “Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks”, by Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson, which appeared in the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019). The dissertation author is the primary investigator and the co-first author of this paper.

This chapter contains material from “SubZero: Zero-Copy IO for Persistent Main Memory File Systems”, by Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson, which appeared in the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2020). The dissertation author is the primary investigator and the first author of this paper.

This chapter contains material from “Blaze: Fast Graph Processing on Fast SSDs”, by Juno Kim and Steven Swanson, which appeared in the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2022). The dissertation author is the primary investigator and the first author of this paper.

This chapter contains material from “TOSS: Tiering of Serverless Snapshots for Low Cost Serverless Computing”, by Juno Kim, Theodoros Michailidis, Linsong Guo, Jishen Zhao, and Steven Swanson, which is in preparation for submission to the European Conference on Computer Systems (EuroSys 2024). The dissertation author is the primary investigator and the co-first author of this paper.

Chapter 3

Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks

Non-volatile main memory (NVMM) technologies like 3D XPoint [74] promise vast improvements in storage performance, but they also upend conventional design principles for the storage stack and the applications that use them. Software designed with conventional design principles in mind is likely to be a poor fit for NVMM due to its extremely low latency (compared to block devices) and its ability to support an enormous number of fine-grained, parallel accesses.

The process of adapting existing storage systems to NVMMs is in its early days, but important progress has been made: Researchers, companies, and open-source communities have built *native NVMM file systems* specifically for NVMMs[31, 36, 106, 112, 113, 63], both Linux and Windows have created *adapted NVMM file systems* by adding support for NVMM to existing file systems (e.g., ext4-DAX, xfs-DAX and NTFS), and some commercial applications have begun to leverage NVMMs to improve performance [10].

System support for NVMM brings a host of potential benefits. The most obvious of these is faster file access via conventional file system interfaces (e.g., `open`, `read`, `write`, and `fsync`). These interfaces should make leveraging NVMM performance easy, and several papers [112, 113, 34, 63] have shown significant performance gains without changing applications, demonstrating the benefits of specialized NVMM file systems.

A second, oft-cited benefit of NVMM is *direct access (DAX)* mmap, which allows an application to map the pages of an NVMM-backed file into its address space and then access it via load and store instructions. DAX removes all of the system software overhead for common-case accesses enabling the fastest-possible access to persistent data. Using DAX requires applications to adopt an mmap-based interface to storage, and recent research shows that performance gains can be significant [30, 107, 85, 73].

Despite this early progress, several important questions remain about how applications can best exploit NVMMs and how file systems can best support those applications. These questions include:

1. How much effort is required to adapt legacy applications to exploit NVMMs? What best practices should developers follow?
2. Are sophisticated NVMM-based data structures necessary to exploit NVMM performance?
3. How effectively can legacy files systems evolve to accommodate NVMMs? What trade-offs are involved?
4. How effectively can current NVMM file systems scale to many-core, multi-socket systems? How can we improve scalability?

This paper offers insight into all of these questions by analyzing the performance and behavior of benchmark suites and full-scale applications on multiple NVMM-aware file systems on a many-core machine. We identify bottlenecks caused by application design, file system algorithms, generic kernel interfaces, and basic limitations of NVMM performance. In each case, we either apply well-known techniques or propose solutions that aim to boost performance while minimizing the burden on the application programmer, thereby easing the transition to NVMM.

Our results offer a broad view of the current landscape of NVMM-optimized system software. Our findings include the following:

- For the applications we examined, FiLe Emulation with DAX (FLEX) provides almost as much benefit as building complex crash-consistent data structures in NVMM.
- Block-based journaling mechanisms are a bottleneck for adapted NVMM file systems. Adding DAX-aware journaling improves performance on many operations.
- The block-based compatibility requirements of adapted NVMM file systems limit their performance on NVMM in some cases, suggesting that native NVMM file systems are likely to maintain a performance advantage.
- Poor performance in accessing non-local memory (NUMA effects) can significantly impact NVMM file system performance. Adding NUMA-aware interfaces to NVMM file systems can relieve these problems.

The remainder of the paper is organized as follows. Section 6.1 describes NVMMs, NVMM file system design issues, and the challenges that NVMM storage stacks face. Section 3.2 evaluates applications on NVMM and recounts the lessons we learned in porting them to NVMMs. Section 3.3 describes the scalability bottlenecks of NVMM file systems and how to fix them, and Section 6.4 concludes.

3.1 Background

This section provides a brief survey of the current state of NVMM-aware file systems, and the challenges of accessing NVMM directly with DAX.

3.1.1 NVMM File Systems and DAX

NVMMs' low latency makes software efficiency much more important than in block-based storage systems [23, 17, 109, 116]. This difference has driven the development of several NVMM-aware file systems [31, 36, 111, 106, 34, 112, 113, 63].

NVMM-aware file systems share two key characteristics: First, they implement direct access (DAX) features. DAX lets the file system avoid using the operating system buffer cache: There is no need to copy data from "disk" into memory since file data is always in memory (i.e., in NVMM). As a side effect, the `mmap()` system call maps the pages that make up a file directly into the application's address space, allowing direct access via loads and stores. We refer to this capability as *DAX-mmap*. One crucial advantage of DAX-mmap is that it allows `msync()` to be implemented in user space by flushing the affected cachelines and issuing a memory barrier. In addition, the `fdatasync()` system call becomes a noop.

One small caveat is that the call to `mmap` must include the recently-added MAP_SYNC flag that ensures that the file is fully allocated and its metadata has been flushed to media. This is necessary because, without MAP_SYNC, in the disk-optimized implementations of `msync` and `mmap` that ext4 and xfs provide, `msync` can sometimes require metadata updates (e.g., to lazily allocate a page).

The second characteristic is that they make different assumptions about the atomicity of updates to storage. Current processors provide 8-byte atomicity for stores to NVMM instead of the sector atomicity that block devices provide.

We divide NVMM file systems into two groups. *Native* NVMM filesystems (or just "native file system") are designed especially for NVMMs. They exploit the byte-addressability of NVMM storage and can dispense with many of the optimizations (and associated complexity) that block-based file systems implement to hide the poor performance of disks.

The first native file system we are aware of is BPFS [31], a copy-on-write file system that introduced short-circuit shadow paging and proposed processor architecture extensions to make NVMM programming more efficient. Intel's PMFS [36], the first NVMM file system released publicly, has scalability issues with large directories and metadata operations.

NOVA [112, 113] is a log-structured file system designed for NVMM. It gives each inode a separate log to ensure scalability, and combines logging, light-weight journaling and copy-on-write to provide strong atomicity guarantees to both metadata and data. NOVA also includes snapshot and fault-tolerance features. NOVA is the only native DAX file system that is publicly available and supported by recent kernels (Intel has deprecated PMFS). It outperforms PMFS on all the workloads for which we have compared them.

Strata [63] is a "cross media" file system that runs partly in userspace. It provides strong atomicity and high performance, but does not support DAX[1].

Adapted NVMM file systems (or just "adapted file systems") are block-based file systems extended to implement NVMM features, like DAX and DAX-mmap. Xfs-DAX [27], ext4-DAX [111] and NTFS [45] all have modes in which they become adapted file systems. Xfs-DAX and ext4-DAX are the state-of-the-art adapted NVMM file systems in the Linux kernel. They add DAX support to the original file systems so that data page accesses bypass the page cache, but metadata updates still go through the old block-based journaling mechanism [25, 26].

So far, adapted file systems have been built subject to constraints that limit how much they can change to support NVMM. For instance, they use the same on-"disk" format in both block-based and DAX modes, and they must continue to implement (or

[1]We have been working to include a quantitative comparison to Strata in this study, but we have run into several bugs and limitations. For example, it has trouble with multi-threaded workloads [62] and many of the workloads we use do not run successfully. Until we can resolve these issues, we have included qualitative discussion of Strata where appropriate.

at least remain compatible with) disk-centric optimizations.

Adapted file systems also often give up some features in DAX mode. For instance, ext4 does not support data journaling in DAX mode, so it cannot provide strong consistency guarantees on file data. Xfs disables many of its data integrity features in DAX mode.

3.1.2 NVMM programming

DAX-mmap gives applications the fastest possible access to stored data and allows them to build complex, persistent, pointer-based data structures. This usage model has the application create a large file in a NVMM file system, use `mmap()` to map it into its address space, and then rely on a userspace persistent object library [30, 107, 85] to manage it.

These libraries generally provide persistent memory allocators, an object model, and support for transactions on persistent objects. To ensure persistence and consistency, these libraries use instructions such as `clflushopt` and `clwb` to flush the dirty cachelines [48, 124] to NVMM, and non-temporal store instructions like `movntq` to bypass the CPU caches and write directly to NVMM. Enforcing ordering between stores requires memory barriers such as `mfence`.

Using mapped NVMM to build complex data structures is a daunting challenge. Programmers must manage concurrency, consistency, and memory allocation all while ensuring that the program can recover from an ill-timed system crash. Even worse, data structure corruption and memory leaks are persistent, so rebooting, the always-reliable solution to volatile data structure corruption and DRAM leaks, will not help. Addressing these challenges is the subject of a growing body of research [30, 107, 85, 105, 117, 73, 12].

3.2 Adapting applications to NVMM

The first applications to use NVMM in production are likely to be legacy applications originally built for block-based storage. Blithely running that code on a new, faster storage system will yield some gains, but fully exploiting NVMM's potential will require some tuning and modification. The amount, kind, and complexity of tuning required will help determine how quickly and how effectively applications can adapt.

We gathered the first-hand experience with porting legacy applications to NVMM-based storage systems by modifying five lightweight databases and key-value stores to better utilize NVMMs. The techniques we applied for each application depend on how it accesses the underlying storage. Below, we detail our experience and identify some best practices for NVMM programmers. Then, based on these findings, we propose a DAX-aware journaling scheme for ext4 that eliminates block IO overheads.

We use a quad-socket prototype HPE Scalable Persistent Memory server [46] to evaluate these applications. The server combines DRAM with NVMe SSDs and an integrated battery backup unit to create NVMM. The server hosts four Xeon Gold 6148 processors (a total of 80 cores), 300 GB of DRAM, and 300 GB of NVMM. We evaluate all the applications on Linux kernel 4.13.

3.2.1 SQLite

SQLite [98] is a lightweight embedded relational database that is popular in mobile systems. SQLite stores data in a B+tree contained in a single file.

To ensure consistency, SQLite uses one of four different techniques to log operations to a separate log file. Three of the techniques, DELETE, TRUNCATE and PERSIST, store undo logs while the last, WAL stores redo logs.

The undo logging modes invalidate the log after every operation. DELETE and TRUNCATE, respectively, delete the log file or truncate it. PERSISTS issues a write to

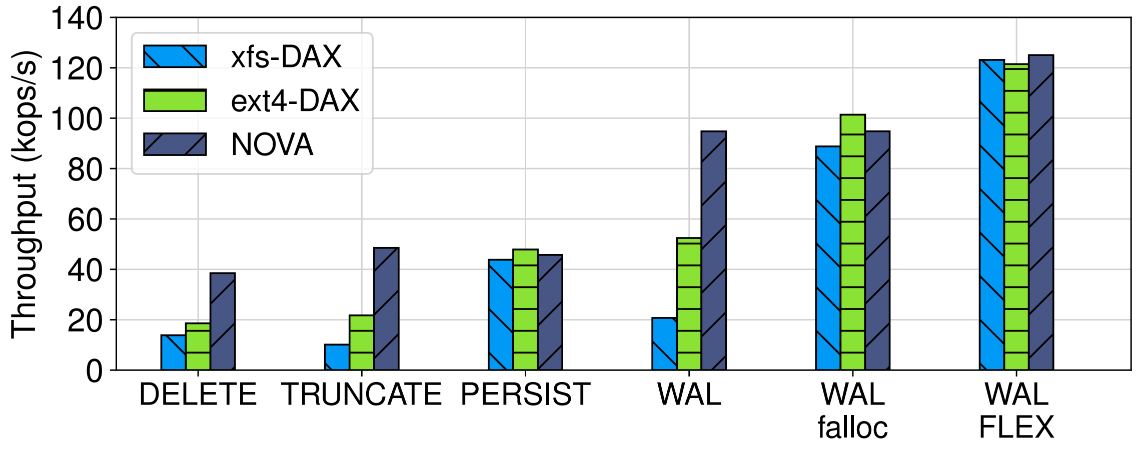


Figure 3.1. SQLite SET throughput with different journaling modes. Preallocating space for the log file using `falloc` avoids allocation overheads and makes write ahead logging (WAL) the clearly superior logging mode for SQLite running on NVMM file systems.

set an "invalid" flag in log file header.

WAL appends redo log entries to a log file and takes periodic checkpoints after which it deletes the log and starts again.

We use Mobibench [50] to test the SET performance of SQLite in each journaling mode. The workload inserts 100 byte long values into a table. Figure 3.1 shows the result. DELETE and TRUNCATE incur significant file system journaling overhead with ext4-DAX and xfs-DAX. NOVA performs better because it does not need to journal operations that affect a single file. PERSIST mode performs equally on all three file systems.

WAL avoids file creation and deletion, but it does require allocating new space for each log entry. Ext4-DAX and xfs-DAX keep their allocator state in NVMM and keep it consistent at all times, so the allocation is expensive. Persistent allocator state is necessary in block-based file systems to avoid a time-consuming (on disk) media scan after a crash.

Scanning NVMM after crash is much less costly, so NOVA keeps allocator state in DRAM and only writes it to NVMM on a clean unmount. As a result, the allocation is much faster.

This difference in allocation overhead limits WAL's performance advantage compared to PERSIST to 9% for ext4-DAX, reduces performance by 53% for xfs-DAX, but improves NOVA's performance by 107%.

We modified SQLite to avoid allocation overhead by using `fallocate` to pre-allocate the WAL file. This is a common optimization for disk-based file systems, and it works here as well: The change closes the gap between the three file systems.

To improve performance further, we use a technique we call FiLe Emulation with DAX (FLEX) to avoid the kernel completely for writes to the WAL file. To implement FLEX, SQLite DAX-mmaps the WAL file into its address space and uses non-temporal stores and `clwb` to ensure the log entries are reliably stored in NVMM. We study FLEX in detail in Section 3.2.4. Implementing these changes required changing just 266 lines of code but improved performance by between 15% and 38%, and further narrows the performance gap between the three file systems.

This final DAX-aware version of SQLite outperforms the PERSIST version by between 2.5× and 2.8×.

Other groups have adapted SQLite to solid-states storage as well. Jeong *et al.* [51] and WALDIO [66] investigate SQLite I/O access patterns and implement optimizations in ext4's journaling system or SQLite itself to reduce the cost of write-ahead logging. Our approach is similar, but it leverages DAX to avoid the file system and leverage NVMM. SQLite/PPL [80], NVWAL [60] use slotted paging [92] to make SQLite run efficiently on NVMM. A comparison to these systems would be interesting, but unfortunately, none of them is publicly available.

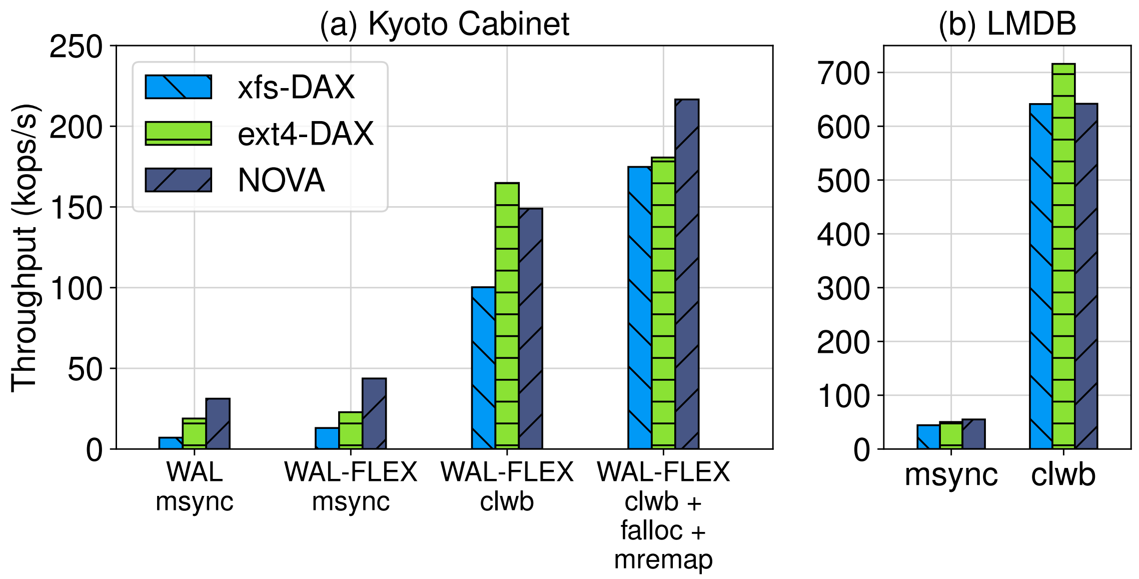


Figure 3.2. Kyoto Cabinet (KC) and LMDB SET throughput. Applications that use `mmap` can improve performance by performing `msync` in userspace.

3.2.2 Kyoto Cabinet and LMDB

Even without DAX, some applications access files via `mmap`, and this makes them a natural match for DAX file systems. However, maximizing the benefits of DAX still requires some changes. We select two applications to explore what is required: Kyoto Cabinet and LMDB.

Kyoto Cabinet Kyoto Cabinet [40] (KC) is a high performance database library. It stores the database in a single file with database metadata at the head. Kyoto Cabinet memory maps the metadata region, uses load/store instructions to access and update it, and calls `msync` to persist the changes. Kyoto Cabinet uses write-ahead logging to provide failure atomicity for SET operations.

Figure 3.2 shows the impact of applying optimizations to KC's database file and its write-ahead log. First, we change KC to use FLEX writes to update the log ("WAL-FLEX msync" in the figure). The left two sets of bars in Figure 3.2 (a) show the impact of these changes. The graph plots throughput for SET operation on Kyoto

Cabinet HashDB. The key size is 8 bytes and value size is 1024 bytes. FLEX write improves performance by 40% for NOVA, 20% for ext4-DAX, and 84% for xfs-DAX.

Kyoto Cabinet calls `msync` frequently on its data file to ensure that updates to memory-mapped data are persistent. DAX-mmap allows userspace to provide these guarantees using a series of `clwb` instructions followed by a memory fence. Flushing in userspace is also more precise since `msync` operates on pages rather than cache lines. Avoiding `msync` improves performance further by 3.4× for NOVA, 7.2× for ext4-DAX, and 7.7× for xfs-DAX ("WAL-FLEX clwb").

By default, Kyoto Cabinet only mmaps the first 64 MB of the file, which includes the header and ∼63 MB of data. It uses `write` to append new records to the file. Our final optimization uses `fallocate` and `mremap` to resize the file ("WAL-FLEX clwb + falloc + mremap"). It boosts the throughput for all the file systems by between 7× to 25×, compared to the baseline implementation that issued `msync` system calls without WAL optimization.

Implementing all of these optimizations for both files required changing just 181 lines of code.

LMDB Lightning Memory-Mapped Database Manager (LMDB) [101] is a Btree-based lightweight database management library. LMDB memory-maps the entire database, so that all data accesses directly load and store the mapped memory region. LMDB performs copy-on-write on data pages to provide atomicity, a technique that requires frequent `msync` calls.

For LMDB, using `clwb` instead of `msync` improves the throughput by between 11× to 14× (Figure 3.2 (b)). Ext4-DAX out-performs xfs-DAX and NOVA by about 11% because ext4-DAX supports super-page (2 MB) `mmap` which reduces the number of page faults. These changes entailed changes to 101 lines of code.

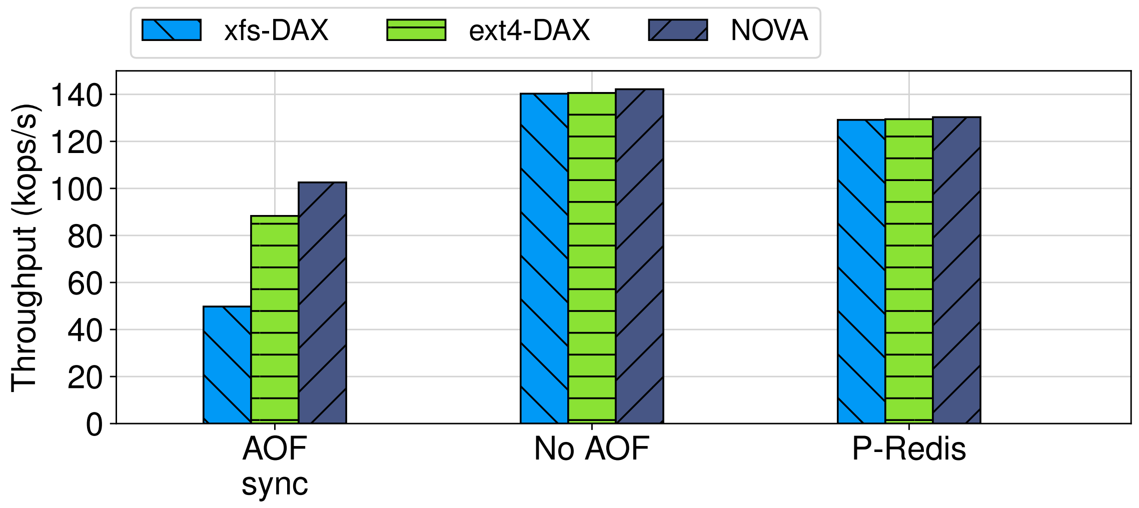


Figure 3.3. Redis MSET throughput. Making Redis' core data structure persistent in NVMM (P-Redis) improves performance by 27% to 2.6×.

3.2.3 RocksDB and Redis

Since disk is slow, many disk-based applications keep data structures in DRAM and flush them to disk only when necessary. To provide persistence, they also record updates in a persistent log, since sequential access is most efficient for disks. We consider two such applications, Redis and RocksDB, to understand how this technique can be adapted to NVMM.

Redis Redis [87] is an in-memory key-value store widely used in web site development as a caching layer and for message queue applications. Redis uses an "append only file" (AOF) to log all the write operations to the storage device. At recovery, it replays the log. The frequency at which Redis flushes the AOF to persistent storage allows the administrator to trade-off between performance and consistency.

Figure 3.3 measures Redis MSET (multiple set) performance. As we have seen with other applications, xfs-DAX's journaling overheads hurt append performance. The graph also shows the potential benefit of eliminating AOF (and giving up persistence): It improves throughput by 2.8×, 59%, and 38% for xfs-DAX, ext4-DAX, and NOVA, respectively.

The hash table Redis uses internally is an attractive target for NVMM conversion, since making it persistent would eliminate the need for the AOF. We created a fully-functional persistent version of the hash table in NVMM using PMDK [85] by adopting a copy-on-write mechanism for atomic updates: To insert/update a key-value pair, we allocate a new pair to store the data, and replace the old data by atomically updating the pointer in the hashtable. We refer to the resulting system as Persistent Redis (P-Redis).

The throughput with our persistent hash table is 27% to 2.6$\times$ better than using synchronous writes to the AOF, and $\sim$9% worse than skipping persistence altogether. Implementing the persistent version of the hash table took 1529 lines of code.

Although Redis is highly-optimized for DRAM, porting it to NVMM is not straightforward and requires large engineering effort. First, Redis represents and stores different objects with different encodings and formats, and P-Redis has to be able to interpret and handle the various types of objects properly. Second, Redis stores virtual addresses in the hashtable, and P-Redis needs to either adjust the addresses upon restart if the virtual address of the `mmap`'d hashtable file has changed, or change the internal hashtable implementation to use offset instead of absolute addresses [**?**]. Neither option is satisfying, and we choose the former solution for simplicity. Third, whenever Redis starts, it uses a random seed for its hashing functions, and P-Redis must make the seeds constant. Fourth, Redis updates the hashtable entry before updating the value, and P-Redis must persist the key-value pair before updating the hashtable entry for consistency. Finally, P-Redis hashtable does not support resizing as it requires journaling mechanism to guarantee consistency.

RocksDB RocksDB [38] is a high-performance embedded key-value store based on log-structured merge trees (LSM-trees). When applications write data to a LSM-tree, RocksDB inserts the data to a skip-list in DRAM, and appends the data to a write-ahead

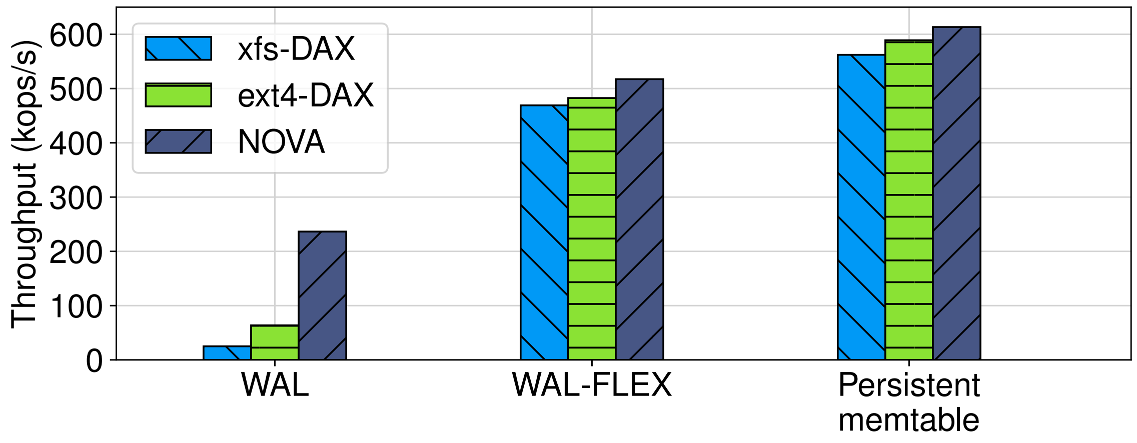


Figure 3.4. RocksDB SET throughput. Appends to the write-ahead log (WAL) file limit RocksDB throughput on NVMM file systems. Using FLEX writes improves performance by 2.2× to 18.7×. Replacing the skip-list and the log with a crash-consistent, persistent skip-list improves throughput by another 19% on average.

log (WAL) file. When the skip-list is full, RocksDB writes it to disk and discards the log file.

Figure 3.4 measures RocksDB SET throughput with 20-byte keys and 100-byte values. RocksDB's default settings perform poorly on xfs-DAX and ext4-DAX, because each append requires journaling for those file systems. NOVA performs better because it avoids this cost.

RocksDB benefits from FLEX as well. It improves throughput by 2.2× - 18.7× and eliminates the performance gap between file systems.

Since the skip-list contains the same information as the WAL file, we eliminate the WAL file by making the skip-list a persistent data structure, similar to NoveLSM [**?**] based on LevelDB. The final bars in Figure 3.4 measure the performance of RocksDB with a crash-consistent skip-list in NVMM. Performance improves by 11× compared to the RocksDB baseline but just 19% compared to optimizing WAL with FLEX.

3.2.4 Evaluating FLEX

In general, FLEX involves replacing conventional file operations with similar DAX-based operations to avoid entering the kernel. We have applied FLEX techniques by hand to the SQLite, RocksDB, and Kyoto Cabinet, but they could easily be encapsulated in a simple library.

FLEX replaces `open()` with `open()` followed by DAX-`mmap()` to map the file into the application's address space. Then, the application can replace `read()` and `write()` system calls with userspace operations.

A FLEX write first checks if the file will grow as a result of the write. If so, the application can expand the file using `fallocate()` and `mmap()` or `mremap()` to expand the mapping. To amortize the cost of `fallocate()`, the application can extend the file by more than the write requires.

Once space is available, a FLEX write uses non-temporal stores to copy data into the file. If the write needs to be synchronous the application issues an `sfence` instruction to ensure the stores have completed. FLEX also uses an `sfence` instruction to replace `fsync()`.

FLEX reads are simpler: They simply translate to `memcpy()`.

FLEX requires the application to track a small amount of extra state about the file, including its location in memory, its current write point, and its current allocated size.

FLEX operations provide semantics that are similar to POSIX, but there are important and potentially subtle differences. First, operations are not atomic. Second, POSIX semantics for shared file descriptors are lost. We have not found these differences to be relevant for the performance-critical file operations in the workloads we have studied. We elaborate this point in Section 3.2.4.

To understand when FLEX improves performance, we constructed a simple

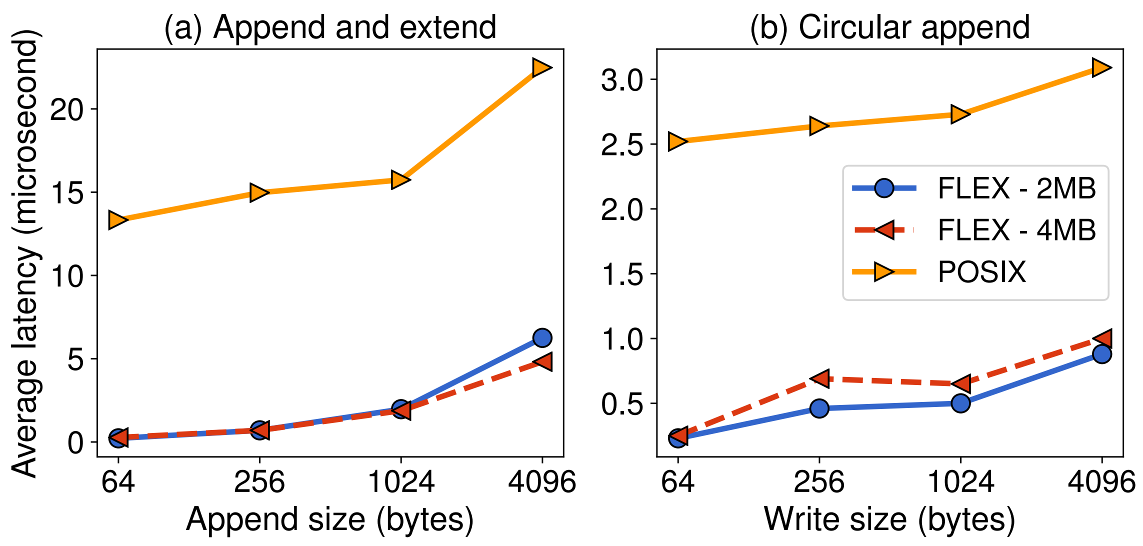


Figure 3.5. The Impact of FLEX File Operations. Emulating file accesses in user space can improve performance for a wide range of access patterns. Note that the Y axes have different scales. "-2 MB" and "-4 MB" denote different `fallocate()` sizes.

microbenchmark that opens a file, and performs a series of reads or writes each followed by `fsync()`. We vary the size and number of operations and the amount of file space we pre-allocate with `fallocate()`. Figure 3.5 shows results for two different cases: "Append and extend" uses FLEX to emulate append operations that always cause the file to grow. "Circular append" reuses the same file area and avoids the need to allocate more space. The applications we studied use both models to implement logging: RocksDB uses "append and extend" whereas SQLite and Kyoto Cabinet use "circular append."

The data show that FLEX outperforms normal write operations by up to 61× for append and extend and up to 11× for circular append. The larger speedup for append and extend is due to the NVMM allocation overhead. Performance gains are especially large for small writes, a common case in the applications we studied.

For use cases that must extend the file, minimizing the cost of space allocation is critical. The results in the figure use 2 MB pages to minimize paging overheads. With 4 KB pages, FLEX only provides speedups for transfers under 4 KB.

Our experience with applying FLEX to RocksDB, SQLite, and Kyoto Cabinet shows that it can provide substantial performance benefits for very little effort. In contrast to re-implementing data structures to be crash-consistent, FLEX requires little to no changes to application logic and requires no additional logging or locking protocols. The only subtleties lie in determining that strict POSIX semantics are not necessary.

These results show that FLEX can provide an easy, incremental, and high-value path for developers creating new applications for NVMM or migrating existing code. It also reduces the importance of using a native NVMM file system, further easing migration, since FLEX performance depends little on the underlying file system.

The Strata file system [63] provides some of the same advantages as FLEX through userspace logging through a library that communicates with the in-kernel file system. Their results show that coupling the user space interface to the underlying file system leads to good performance. Their interface makes strong atomicity guarantees while FLEX lets the application enforce the semantics it requires.

Correctness

Since FLEX is not atomic, applying it to applications that assume atomic writes is likely to cause a correctness problem. To our knowledge, SQLite, RocksDB, and Kyoto Cabinet do not assume the atomicity of write system calls [84], thereby applying FLEX does not break their application logic. Only LMDB assumes that 512 bytes sector writes are atomic [**?**]. Therefore, running it on NVMM file systems introduces the correctness problem since only 8 bytes are atomic on NVMM. To solve this problem, we added a checksum for the LMDB metadata: When a checksum error is detected, LMDB falls back to the previous header.

Table 3.1. Application Optimization Summary The applications we studied used a variety of techniques to reliably store persistent state. All the optimizations we applied improved performance, but the amount of programmer effort varied widely. The data Figures 3.1, 3.2, 3.3, and 3.4 show that programmer effort does not correlate with performance gains.

	Native techniques		Optimizations (Lines changed)		
	WAL	mmap+msync	FLEX	CLWB+fence	Persistent Objects
SQLite	×	-	266	-	-
Kyoto Cabinet	×	×	133	48	-
LMDB	-	×	-	101	-
Redis	×	-	-	-	1326
RocksDB	×	-	56	-	380

3.2.5 Best Practices

Based on our experiences with these five applications, we can draw some useful conclusions about how applications can profitably exploit NVMMs.

Use FLEX Emulating file operations in user space provides large performance gains for very little programmer effort.

Use fine-grained cache flushing instead of `msync` Applications that already use `mmap` and `msync` to access data and ensure consistency, can improve performance significantly by flushing cache lines rather than `msync`'ing pages. However, ensuring that all updated cache lines are flushed correctly can be a challenge.

Use complex persistent data structure judiciously For both of the DRAM data structures we made persistent, the programming effort required was significant and likely performance gains were relatively small relative to FLEX. This finding leads us to two conclusions: First, it is critical to make building persistent data structures in NVMM as easy as possible. Second, it is wise to estimate the potential performance impact the persistent data structure will have before investing a large amount of programmer effort

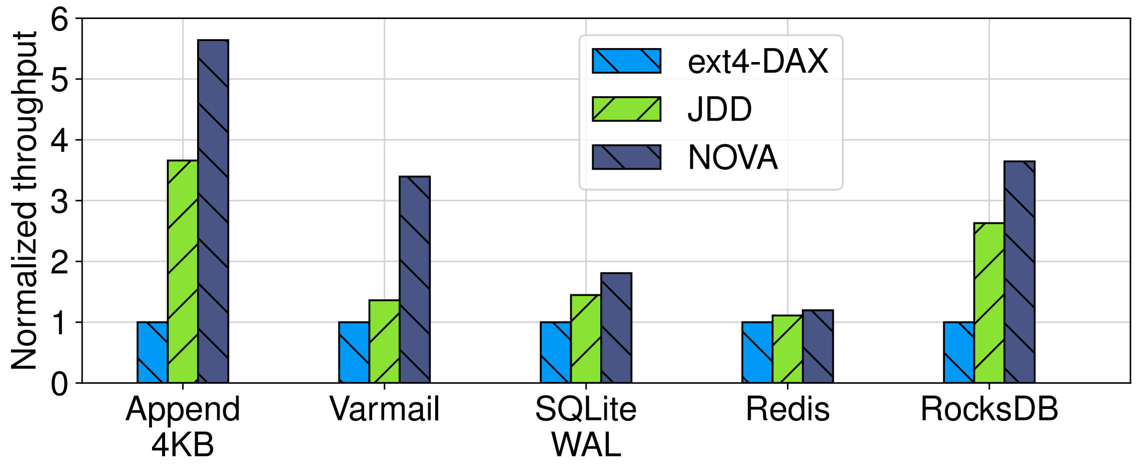


Figure 3.6. JDD performance. Fine-grained, DAX-optimized journaling on NVMM improves performance for metadata-intensive applications.

in developing it [70].

Preallocate files on adapted NVMM file systems Several of the performance problems we found with adapted NVMM file systems stemmed from storage allocation overheads. Using `fallocate` to pre-allocate file space eliminated them.

Avoid meta-data operations Directory operations (e.g., deleting files) and storage allocation incurred journaling overheads in both xfs and ext4. Avoiding them improves performance, but this is not always possible.

3.2.6 Reducing journaling overhead

Several of the best practices we identify above focus on avoiding metadata operations since they are often slow. This can be awkward and some metadata operations are unavoidable, so improving their performance would make adapting to NVMMs easier and improve performance.

NOVA's mechanism for performing consistent metadata updates is tailored specifically for NVMMs, but ext4 and xfs' journaling mechanisms were built for disk, and this legacy is evident in their poorer metadata performance.

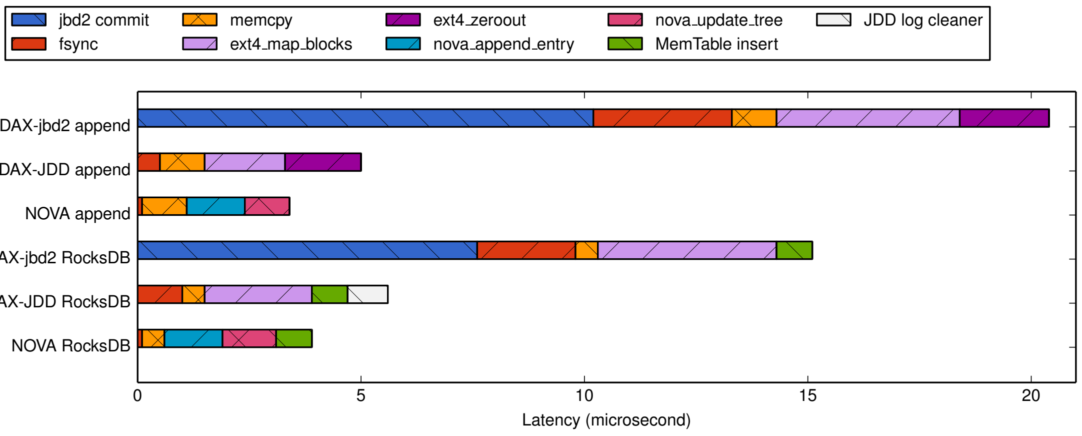


Figure 3.7. Latency break for 4KB append and RocksDB SET. JDD significantly reduces journaling overhead by eliminating JBD2 transaction commit, but still has higher latency than NOVA's metadata update mechanism.

Ext4 uses the journaling block device (JBD2) to perform consistent metadata updates. To ensure atomicity, it always writes entire 4 KB pages, even if the metadata change affects a single byte. Transactions often involve multiple metadata pages. For instance, appending 4 KB data to a file and then calling `fsync` writes one data page and eight journal pages: a header, a commit block, and up to six pages for inode, inode bitmap, and allocator.

JDB2 also allows no concurrency between journaled operations, so concurrent threads must synchronize to join the same running transaction, making the journaling a scalability bottleneck [97]. Son *et al.* [97] and iJournaling [83] have tried to fix ext4's scalability issues by reducing lock contention and adding per-core journal areas to JBD2.

Previous works [25, 26] has identified the inefficiencies of coarse-grain logging and proposed solutions in the context of block-based file systems. FSMAC [26] maintains data in disk/SSD and metadata in NVMM, and uses undo log journaling for metadata consistency. The work in [25] journals redo log records of individual metadata fields to NVMM during transaction commit, and applies them to storage during checkpointing.

To understand how much of ext4's poor metadata performance is due to coarse-grain logging, we apply these fine-grain logging techniques to develop a journaling DAX device (JDD) for ext4 which performs DAX-style journaling on NVMM and provides improved scalability.

JDD makes three key improvements to JBD2. First, it journals individual metadata fields rather than entire pages. Second, it provides pre-allocated, per-CPU journaling areas so CPUs can perform journaled operations in parallel. Third, it uses undo logging in the journals: It copies the old values into the journal and performs updates directly to the metadata structures in NVMM. To commit an update it marks the journal as invalid. During recovery from a crash, the file system rolls back partial updates using the journaled data. These changes provide for very lightweight transaction commit and make checkpointing unnecessary.

JDD differs from the previous works by focusing on NVMM file systems. FSMAC aims to accelerate metadata updates for disk-based file systems by putting the metadata separately in NVMM. To handle the performance gap between NVMM and disk, FSMAC maintains multiple versions of metadata. The work in [25] optimizes ext4 using fine-grained redo logging on NVMM journal. We built JDD to improve the performance of adapted NVMM file systems using fine-grained undo logging, avoiding the complexity of previous works – managing versions in FSMAC or transaction committing and checkpointing in [25].

Strata [63] and Aerie [106] take a more aggressive approach and log updates in userspace under the control of file system-specific libraries. Metadata updates occur later and off the critical path. This approach should offer better performance than the techniques described above since it avoids entering the kernel for metadata updates. However, it also involves more extensive changes to the application.

Figure 3.6 shows JDD's impact on a microbenchmark that performs random 4 KB writes followed by `fsync`, Filebench [102] Varmail (which is metadata-intensive), and the three databases and key value stores we evaluated earlier that perform frequent metadata operations as part of WAL. The JDD improves the microbenchmark performance by 3.7× and varmail by 40%. For applications that use write-ahead logging, the benefits range from 11% to 2.6×.

We further analyze the latency of JDD for 4 KB appends and RocksDB SET operation and show the latency breakdown in Figure 3.7. In ext4-DAX, JBD2 transaction commit (jbd2_commit) occupies 50% of the total latency. JDD eliminates this overhead by performing undo logging. JDD also reduces ext4 overheads such as block allocation (ext4_map_blocks). The remaining performance gap between ext4 and NOVA (46%) is due to ext4's more complex design and its need to keep more persistent states in storage media. In particular (as discussed in Section 3.2.1) ext4 keeps its data block and inode allocator state continually up-to-date on disk.

The performance improvement on Redis and SQLite are smaller, because they have higher internal overheads. Redis spends most of its time on TCP transfers between the Redis server and the benchmark application, and SQLite spends over 40% of execution time parsing SQL and performing B-tree operations.

3.3 File System Scalability

We expect NVMM file systems to be subject to more onerous scalability demands than block-based filesystems due to the higher performance of the underlying media and the large amount of parallelism that modern memory hierarchies can support [18]. Further, since NVMMs attach to the CPU memory bus, the capacity of NVMM file systems will tend to scale with the number sockets (and cores) in the systems.

Many-core scalability is also a concern for conventional block-based file systems, and researchers have proposed potential solutions. SpanFS [55] shards file and directories across cores at a coarse granularity, requiring developers to distribute the files and directories carefully. ScaleFS [18] decouples the in-memory file system from the on-disk file system, and uses per-core operation logs to achieve high concurrency. ScaleFS was built on xv6, a research prototype kernel, which makes impossible to perform a good head-to-head comparison with our changes. However, we expect that applying its techniques and the Scalable Commutativity Rule [29] systematically to NVMM file systems (and the VFS layer) might yield further scaling improvements.

This section first describes the FxMark [76] benchmark suite. Then, we identify several operations that have scalability limitations and propose solutions.

3.3.1 FxMark scalability test suite

Min *et al.* [76] built a file system scalability test suite called FxMark and used it to identify many scalability problems in both file systems and Linux's VFS layer. It includes nineteen tests of performance for data and metadata operations under varying levels of contention.

Min *et al.* use FxMark to identify scalability bottlenecks across many file systems. Interestingly, it is their analysis of tmpfs, a DRAM-based pseudo-file system that reveals the bottlenecks that are most critical for ext4-DAX, xfs-DAX, and/or NOVA.

We repeat their experiments and then develop solutions to improve scalability. The solutions we identify are sufficient to give good scalability with NVMM, but would probably also help disk-based file systems too.

FxMark includes nineteen workloads. Below, we only discuss those that show poor scalability for at least one the NVMM file systems we consider.

3.3.2 Concurrent file read/write

Concurrent `read` and `write` operations to a shared file are a well-known sore spot in file system performance. Figure 3.8 shows scalability problems for both reads and writes across ext4-DAX, xfs-DAX, and NOVA. The root cause of this poor performance is Linux's read/write semaphore implementation [19, 20, 57, 68]: It is not scalable because

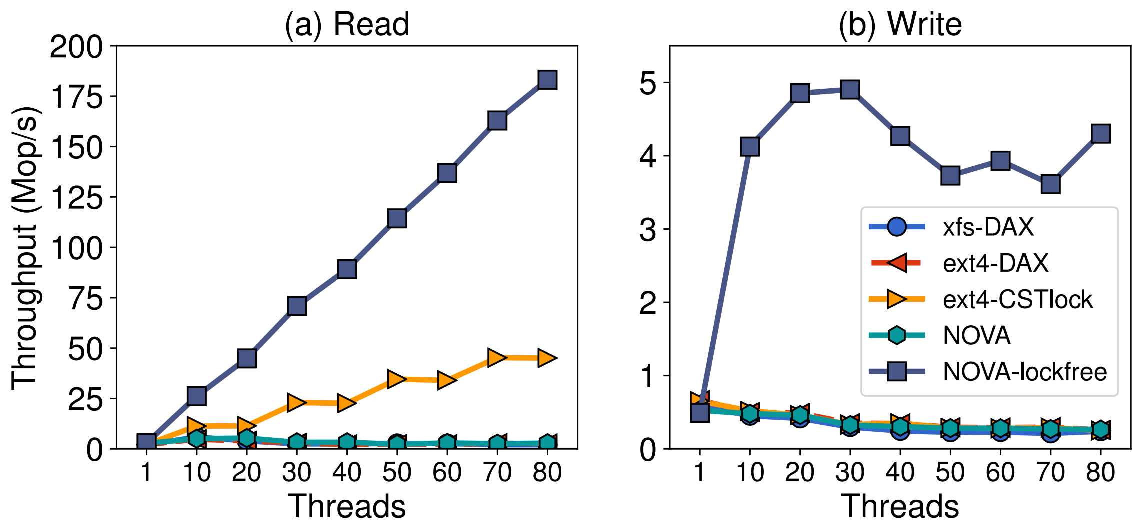


Figure 3.8. Concurrent 4KB read and write throughput. By default, Linux uses a non-scalable reader/writer lock to coordinate access to files. Using finer-grain, more scalable locks improves read and write scalability.

of the atomic update required to acquire and release it.

The semaphore protects two things: The file data and the metadata that describes the file layout. To remove this bottleneck in NOVA, we use separate mechanisms to protect the data and metadata.

To protect file data, we leverage NOVA's logs. NOVA maintains one log per inode. Many of the log entries correspond to write operations and hold pointers to the file pages that contain the data for the write. Rather than locking the whole inode, we use reader/writer locks on each log entry to protect the pages to which it links. Although this lock resides in NVMM, its state is not necessary for recovery and is cleared before use after a restart, so hot locks will reside in processor caches and not usually be subject to NVMM access latency.

NOVA's approach to tracking file layout makes protecting it simple. NOVA uses an in-DRAM radix tree to map file offsets to write entries in the log. Write operations update the tree and both reads and writes query it. Instead of using a lock we leverage the Linux radix tree implementation that uses read-copy update [71] to provide more scalable, concurrent access to a file.

Figure 3.8 shows the results (labeled "NOVA-lockfree") on our 80-core machine. 4 KB read performance scales from 2.9 Mops/s for one thread to 183 Mops/s with 80 threads (63×). The changes improve write performance as well, but write bandwidth saturates at twenty threads because our NVMM is attached to one of four NUMA nodes and each node has twenty threads.

Adding fine-grain locking for ranges of a file is possible for ext4-DAX and xfs-DAX, and it would improve performance when running on any storage device.

Using the radix tree to store file layout information would be more challenging since ext4 and xfs make updates to file layout information immediately persistent in the file's inode and indirect blocks. This is necessary to avoid reading the data from disk when the file is opened, which would be slow on block device. Since NVMM is much faster, NOVA can afford to scan the inode's log on `open` to construct the radix tree in DRAM.

An alternative solution for ext4 and xfs would be to replace VFS's per-inode reader/write semaphore with a CST semaphore [57] (or some other more scalable semaphore). The ext4-CSTlock line in the figure shows the impact on ext4-DAX: Performance scales from 2.1 Mops/s for one thread to 45 Mops/s for eighty threads (21×). The gains are not as large as the approach we implemented in NOVA, and they only apply to reads. Both of these approaches could coexist.

3.3.3 Directory Accesses

Scalable directory operations are critical in multi-program, data intensive workloads. Figure 3.9 shows that creating files in private directories only scales to twenty cores. Min *et al.* identify the root cause, but do not offer a solution: VFS takes a spinlock to add the new inode to the superblock's inode list and a global inode cache. The inode list includes all live inodes, and the inode cache provides a mapping from inode number

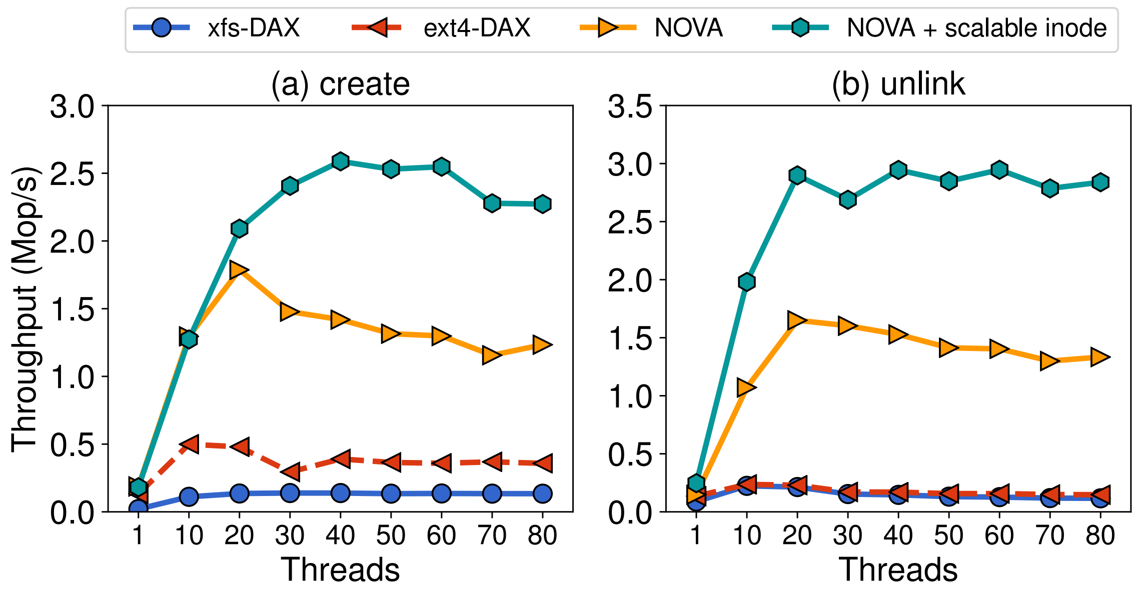


Figure 3.9. Concurrent `create` and `unlink` throughput. The `create` and `unlink` operations are not scalable even if performed in isolated directories, because Linux protects the global inode lists and inode cache with a single spinlock. Moving to per-cpu structures and fine-grain locks improves scalability above 20 cores.

to inode addresses.

We solve this problem and improve scalability for the inode list by breaking it into per-CPU lists and protecting each with a private lock. The global inode cache is an open-chaining hash table with 1,048,576 slots. We modify NOVA to use a per-core inode cache table. The table is distributed across the cores, each core maintains a radix tree that provides lock-free lookups, and threads on different cores can perform inserts concurrently. In Figure 3.9, the "NOVA + scalable inode" line shows the resulting improvements in scaling.

Updates to shared directories also scale poorly due to VFS locking. For every directory operation, VFS takes the inode mutexes of all the affected inodes, so operations in the shared directories are serialized. The `rename` operation is globally serialized at a system level in the Linux kernel for consistent updates of the dentry cache. Fixing these problems is beyond the scope of this paper, but recent work has addressed

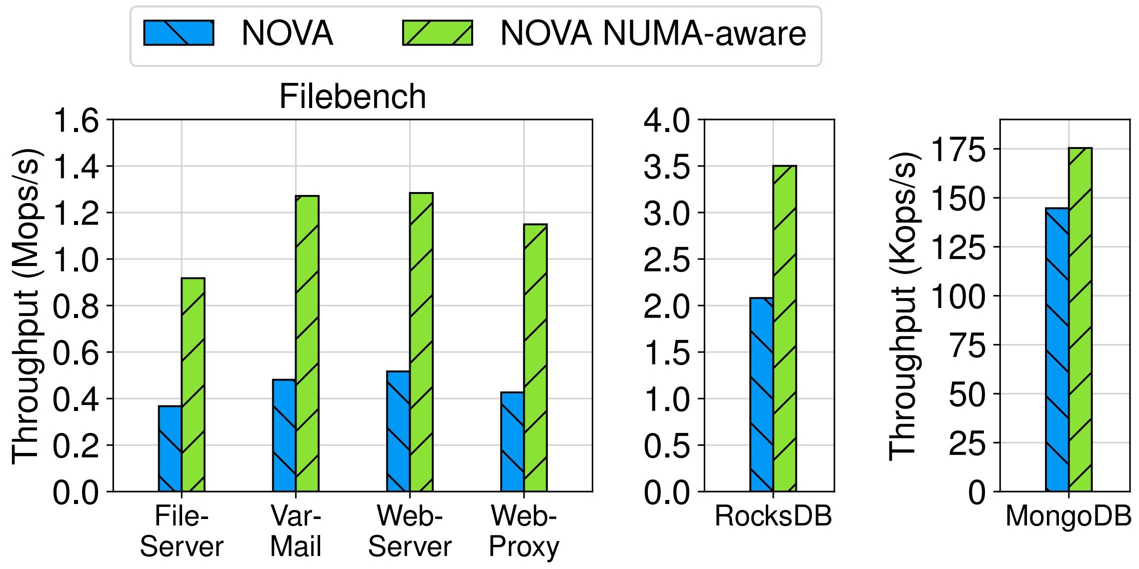


Figure 3.10. NUMA-awareness in the file system. Since NVMM is memory, NUMA effects impact performance. Providing simple controls over where the file system allocates NVMM for a file lets application run threads near the data they operate on, leading to higher performance.

them [103, 18].

3.3.4 NUMA Scalability

Intelligently allocating memory in NUMA systems is critical to maximizing performance. Since a key task of NVMM file systems is allocating memory, these file systems should be NUMA-aware. Otherwise, poor data placement decisions will lead to poor performance [35].

We have added NUMA-aware features to NOVA to understand the impact they can have. We created a new `ioctl` that can set and query the preferred NUMA node for the file. A NUMA node represents a set of processors and memory regions that are close to one another in terms of memory access latency. The file system will try to use that node to allocate all the metadata and data pages for that file. A thread can use this `ioctl` along with Linux's CPU affinity mechanism to bind itself to the NUMA node where the file's data and metadata reside.

Figure 3.10(left) shows the result of Filebench workloads running with fifty threads. The NVMM is attached to NUMA node 0. Without the new mechanism, threads are spread across all the NUMA nodes, and some of them are accessing NVMM remotely. Binding threads to the NUMA node that holds the file they are accessing improves performance by 2.6× on average.

The other two graphs in Figure 3.10 measure the impact on RocksDB and MongoDB [77]. We modified RocksDB to schedule threads on the same NUMA node as the `SSTable` files using our `ioctl`, and ran `db_bench readrandom` benchmark with twenty threads. Similarly, we modified MongoDB to enable NUMA-aware thread scheduling, and ran read-intensive (95% read, 5% update) YCSB benchmark [32] with twenty threads. For both workloads, the data set size is 30 GB. The graphs show the result: NUMA-aware scheduling improves RocksDB and MongoDB performance by 68% and 21%, respectively.

3.4 Summary

We have examined the performance of NVMM storage software stacks to identify the bottlenecks and understand how both applications and the operating system should adapt to exploit NVMM performance.

We examined several applications and identified several simple techniques that provide significant gains. The most widely applicable of these use FLEX to move writes to user space, but implementing `msync` in userspace and assiduously avoiding metadata operations also help, especially on adapted NVMM file systems. Notably, our results show that FLEX can deliver nearly the same level of performance as building crash-consistent data structures in NVMM but with much less effort.

On the file system side, we evaluated solutions to the problems of inefficient logging in adapted NVMM file systems, multicore scaling limitations in file systems and the Linux's VFS layer, and the novel challenge of dealing with NUMA effects in the context of NVMM storage.

Overall, we find that although there are many opportunities for further improvement, the efforts of systems designers over the last several years to prepare systems for NVMM have been largely successful. As a result, there are a range of attractive paths for legacy applications to follow as they migrate to NVMM.

Acknowledgements

This chapter contains material from "Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks", by Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson, which appeared in the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019). The dissertation author is the primary investigator and the co-first author of this paper.

Chapter 4

SubZero: Zero-Copy IO for Persistent Main Memory File Systems

POSIX `read()` and `write()` have been the most common interface for accessing file contents for many years and across many generations of storage hardware. The semantics of these system calls rely crucially on copying data between (volatile) memory and a storage medium (e.g., a disk).

Copy-based semantics are a natural fit for both the performance characteristics and hardware interface of conventional storage technologies. Disks are slow enough that the overhead of the copy is not significant. And even if disks were fast, processors cannot operate directly on the data they hold, because they are not memory.

The copy-based, atomic semantics that `read()` and `write()` provide are also convenient for the programmer. The copy that `read()` creates will not change if the file it reads is overwritten, and `write()` atomically transfers a fully-prepared buffer of data into the file.

The appearance of fast, persistent memory (PMEM) that resides on the processor's memory bus, however, upends both of these long-standing assumptions: PMEM is fast enough that the overhead of a copy is detrimental to performance and the copy is not necessary since the data is already directly-accessible to the processor.

Despite the costs of copy-based operations and because of their convenience and ubiquity, even file systems specifically designed for PMEM [31, 36, 106, 112, 113, 63, 114] implement POSIX-compliant `read()` and `write()`. That said, PMEM file systems generally also provide direct-access (DAX) `mmap()` as an alternative that dispenses with the copy overhead. However, DAX-`mmap()` forces the programmer to implement atomicity and concurrency control manually, complicating the programming model. Moreover, its unclear interaction with `read()` and `write()` has discouraged them from taking the potentially interesting path that leverages both interfaces in a single program.

To bridge the gap between POSIX `read()` and `write()` and DAX `mmap()`, we propose a new IO interface called SubZero that does not rely on copy-based semantics for data access but still preserves the ease of use that `read()` and `write()` provide. SubZero offers DAX-like speed with a simple, POSIX-like interface that interacts cleanly with the legacy POSIX interface.

Concretely, SubZero provides two new system calls – `peek()` and `patch()` – that give programs access to file data and interoperate cleanly with `read()` and `write()` system calls. The `peek()` system call returns the virtual address of a memory region holding a snapshot of requested file contents. The snapshot is atomic with respect to other file operations and its contents do not change if the underlying file changes. The `patch()` system call takes a pointer to a memory region containing new data and atomically incorporates that region into the target file. `patch()` causes the memory region to become read-only.

A PMEM file system can implement `peek()` by simply manipulating the program's page table. It can implement `patch()` by adjusting the file's layout to incorporate the patched pages, as long as the pages are in persistent memory. Benefiting from `peek()` and `patch()` requires changes to how the application accesses file data and how it allocates and disposes of IO buffers.

We implemented SubZero in two of the state-of-the-art PMEM file systems,

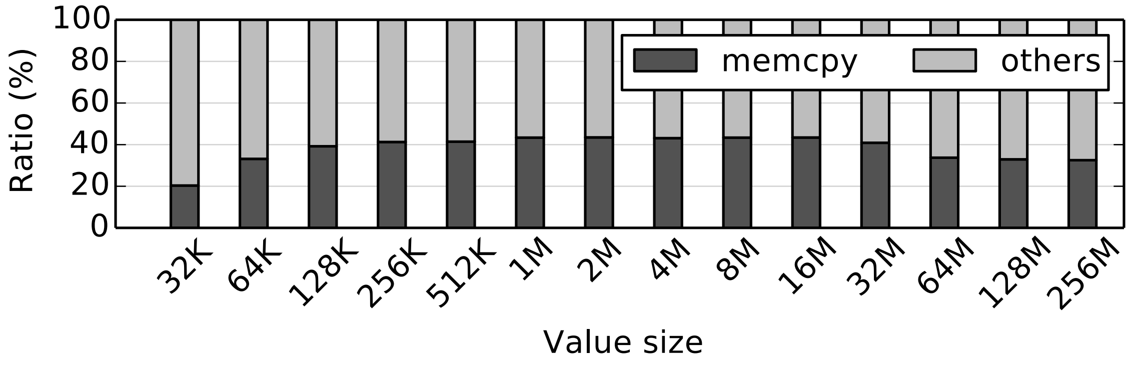


Figure 4.1. Memory copy overhead in Kyoto Cabinet. Memory copy overheads are significant when updating large key-value pairs in Kyoto Cabinet. "others" means application and file system level overheads except for the memory copy.

XFS-DAX and NOVA. Our measurements show that SubZero outperforms copy-based `read()` and `write()` by up to 2× and 6×, respectively. At the application level, `peek()` improves GET performance of the Apache Web Server by 3.6×, and `patch()` boosts the SET performance of Kyoto Cabinet up to 1.3× with non-invasive changes.

The remainder of the paper is organized as follows. Section 4.1 describes the motivation of our work. Section 4.2 describes the details of SubZero IO interface and semantics. Section 4.3 discusses the key implementation details and Section 6.3 evaluates these techniques with micro-benchmarks and applications. Section 5.5 discusses related works and Section 6.4 concludes.

4.1 Motivation

Although conventional POSIX `read()` and `write()` interfaces have proven easy-to-use, they become a major source of inefficiency in PMEM file systems since the media latency of PMEM is low relative to the software overheads on top of PMEM. In the PMEM file system software stack, the copies inherent in the semantics of `read()` and `write()` system calls are a major source of inefficiency.

For example, a simple data copying step between the user buffer and PMEM pages during `write()` system call takes 20 – 45% of the total execution time when updating large key-value pairs in Kyoto Cabinet database [40] backed by the NOVA file system (Figure 4.1). Critically, this copying overhead is a property of the interface, not the file system — all PMEM file systems that implement POSIX IO incur this performance penalty.

PMEM file systems provide DAX-`mmap()` as an alternative to avoid this overhead. However, this interface jettisons the good with the bad. Although it eliminates all common-case file system overheads for data access, it also forces the application to manage crash consistency and concurrency control on its own, complicating the programming model and inviting bugs.

As a solution, SUBZERO bridges the gap between POSIX IO and DAX IO by combining the advantages of both, offering DAX-like speed with a simple, POSIX-like interface that interacts cleanly with itself and the legacy POSIX interface.

4.2 SUBZERO IO

SUBZERO IO (or just SUBZERO) is a suite of new system calls that avoids copy-based semantics and allows for more efficient data access and modification in PMEM-based file systems. SUBZERO strives to provide simple semantics that are easy for programmers to reason about when building sophisticated applications. To these ends, SUBZERO has the following design goals:

- **Zero data movement** SUBZERO should not require costly data movements to implement read and write operations. In particular, SUBZERO performs *no* data movement while the conventional "zero-copy" IO (i.e., conducting IO with `O_DIRECT`) in disk-based file systems still requires one data movement between the storage media and memory.

- **Atomicity** SUBZERO should provide atomicity guarantees similar to those for

Table 4.1. SUBZERO IO functions. The API includes replacements for conventional `read()` and `write()`, in addition to ancillary functions for allocating PMEM buffers for IO. Only `peek()`, `unpeek()`, and `patch()` need to be system calls.

	Function	Semantics
Read	`void* peek(int fd, off_t pos, size_t len)`	Open a read-only mapping to the target range of a PMEM file.
	`int unpeek(void *addr)`	Close a mapping opened by `peek()`.
Write	`int patch(int fd, void *buf, size_t len, off_t pos)`	Update a target file with a PMEM buffer. `patch()` returns an error when `buf` is misaligned with `pos` or the buffer and the target file do not belong to the same file system.
	`int create_pmem_pool(char *path, size_t size)`	Create a PMEM pool from which PMEM pages are allocated and return a unique pool id.
	`void delete_pmem_pool(int pool_id)`	Delete a PMEM pool.
	`void* alloc_pmem(int pool_id, off_t pos, size_t len)`	Allocate a PMEM buffer. `pos` is the target location where the buffer will be patched.
	`void free_pmem(void* buf)`	Reclaim and unmap a PMEM buffer. PMEM pages are reclaimed to the allocator only when they do not belong to files.

POSIX `read()` and `write()`, since those guarantees have proven useful in building IO-intensive applications.

- **Clean integration with POSIX** Combined with their atomicity, SubZero should integrate cleanly with POSIX `read()` and `write()` operations. This allows programmers to freely intermingle those conventional IO operations with SubZero operations.

Below, we describe the SubZero IO interface at a high level and discuss its semantics in more detail. Then we present an example of its use and discuss the changes it requires to existing programs.

4.2.1 The SubZero Interface

SubZero introduces two new IO operations: `peek()` and `patch()`. Table 4.1 summarizes the SubZero interface, which includes these two and several ancillary functions. Below we describe the interface in detail.

`peek()` The `peek()` system call returns a pointer to a memory region that contains the contents of a file at a particular file offset. The region reflects a snapshot of the file contents at the time `peek()` is executed. The snapshot is atomic with respect to file modifications (e.g., `write()` or `patch()`). The snapshot is immutable, so attempts to alter its contents result in a segmentation fault. There are no alignment restrictions (or guarantees) on the file offset or the returned pointer.

Since `peek()` allocates the memory region containing the snapshot, the application must eventually release the memory by passing the snapshot address to `unpeek()`.

Lines 1–3 in Figure 4.2 illustrate how to `peek()` an entire file. Line 9 deallocates the resulting buffer with `unpeek()`.

The `peek()` system call resembles DAX-`mmap()`, since both map file contents into the user address space. However, there are two key differences. First, `peek()`

does not impose any alignment restrictions on the file offset of the region to be peeked, while `mmap()` requires the offset to a multiple of the file system page size. Similarly, `unpeek()` relaxes `munmap()`'s alignment restriction. Second, `peek()` is easier to use than `mmap()`, since the snapshot is explicitly atomic relative to other file modifications and immutable.

`patch()` The `patch()` system call modifies a file by merging the contents of a buffer into a file at the given offset. In essence, the buffer *becomes* part of the file rather than being copied into it. After the `patch()`, the buffer becomes immutable. The state of the `patch()`'d buffer is identical to the state of a `peek()`'d buffer: both are immutable mappings of a file's contents. The `patch()`'d buffer is closed by calling `free_pmem()`.

The change that `patch()` makes to the file is atomic with respect to other `patch()` and `write()` operations. Like `write()`, its effects are not guaranteed to be permanent until the program calls `fsync()` or `fdatasync()`.

The benefit of `patch()` is realized when two conditions are satisfied. First, the buffer should comprise PMEM that is managed by the same file system instance as the file being written to. A program can acquire such PMEM by creating a temporary file in the same file system and `mmap()`ing it. Second, the buffer should be *access-aligned*. Intuitively, this means the page boundaries of the buffer must align with the page boundaries in the file. That is, for a `patch()` operation using a buffer `B` and a file offset `off` on a file system with page size `S`, the `patch()` is access-aligned if `B % S == off % S`. The requirement is similar to the alignment requirement imposed by opening a file with `O_DIRECT`.

To make it easy for a program to satisfy these criteria, we have implemented a simple allocator to allocate and deallocate access-aligned buffers. Program can create a pool in a file system and use it to allocate buffers for `patch()` operations.

Lines 4-8 of Figure 4.2 demonstrate how `patch()` can work with `peek()` to

```
 1 int in_fd = open("/mnt/in.uu", O_RDONLY);                      // Open the
       input file
 2 int input_length = lseek(in_fd, 0L, SEEK_END);                 // Find its
       length
 3 char *in_buf = peek(in_fd, 0, input_length);                   // Peek its
       contents
 4 int pool_id = create_pmem_pool("/mnt", 1073741824);            // Create a
       pool
 5 void *out_buf = alloc_pmem(pool_id, 0, input_length);          // Allocate an
        access-aligned buffer
 6 int output_length = uudecode(in_buf, out_buf, input_length); // Construct
       uudecoded output in the buffer
 7 int out_fd = open("/mnt/out.dat", O_WRONLY);                   // Open the
       output file
 8 patch(out_fd, out_buf, output_length, 0);                      // Patch the
       uudecoded output into the file
 9 unpeek(in_buf);                                                // Unpeek the
       input
10 free_pmem(out_buf);                                            // Unmap the
       output buffer
11 delete_pmem_pool(pool_id);                                     // Delete the
       pool
```

Figure 4.2. Computation between two PMEM files without any copies using SUBZERO. An application can use `peek()` and `patch()` to access and update files without any copies. Here, `uudecode()` reads directly from the input file's pages and writes its result directly to the physical pages that will become the output file.

avoid any unnecessary copies. The code allocates a PMEM buffer and calls `uudecode()` to populate it by processing the `peek()`'d contents of the input. The `patch()` on line 8 causes the output buffer to become the contents of the file, and therefore no copies are necessary.

4.2.2 Using SUBZERO

SUBZERO removes the implicit copy from the semantics of accessing and modifying a file's contents. Realizing its benefits will require changes to how applications perform IO.

The most significant changes affect how the program allocates and manages IO buffers. First, the program can no longer pre-allocate read buffers, since `peek()`

allocates and returns a populated buffer. The program will also need to call `unpeek()` when it is finished with the buffer. Second, the application needs to allocate write buffers from PMEM.

How invasive this change is will depend on how the application performs writes. High-performance applications that maintain their own page-aligned buffer pools and leverage `O_DIRECT` will have less trouble since page-aligned buffers are automatically access-aligned. Applications that perform more ad-hoc writes will need to allocate an access-aligned buffer for each write.

4.3 Implementing SUBZERO

To illustrate its potential, we implemented SUBZERO in two state-of-the-art, in-kernel PMEM file systems: NOVA [112], a file system built from scratch for PMEM, and XFS-DAX, a linux file system adapted to accomodate direct access to PMEM. SUBZERO can be implemented without invasive changes if the file system has the ability to 1) allow multiple files to share data pages and 2) support copy-on-write updates when a write modifies shared pages.

4.3.1 NOVA file system

NOVA [112] is a log-structured file system for persistent memory. It manages a contiguous region of PMEM and presents a POSIX-compatible file system interface. It supports DAX-style `mmap()`, and all file and directory operations are atomic.

NOVA stores per-inode log that contains *write entries* pointing to data pages and describing the file's layout. To perform a write, NOVA allocates new pages, populates them, and then appends a write entry to the log incorporating the pages into the file. Some old pages may become obsolete as a result — NOVA reclaims these during garbage collection. Log append is atomic, so writes are atomic as well. On file open, NOVA scans the log and builds an in-DRAM index that maps file offsets to physical pages.

`peek()` in NOVA The implementation of `peek()` in NOVA mirrors its implementation of `mmap()` with a few additions. Since `peek()` maps the target pages of the file into the application's address space, the implementation must protect against two types of unexpected changes to the `peek()`'d data. First, it must prevent stores by the application from altering the underlying file. Second, it must prevent changes to the file from altering the data visible to the program.

To address the first, NOVA maps the pages as read-only so that attempts to alter the contents from the `peek()`'d address will see the segmentation fault.

To address the second, NOVA does not modify data in place, so the PMEM pages that `peek()` mapped remain unchanged even if the file's contents change. NOVA must take care, however, to prevent the mapped pages from being reclaimed by garbage collection until they are `unpeek()`'d.

`patch()` in NOVA Patch allows programs to insert populated buffers of data directly into a file. This requires solving two problems.

The first is implementing `alloc_pmem()` to allocate buffers in PMEM that are suitable for being `patch()`'d into the file. We solve this by building a userspace library that creates temporary files in PMEM and maps them with DAX-`mmap()` to return pointers to programs that call `alloc_pmem()`.

The second is performing the `patch()` itself. In NOVA, `patch()` is a special case of a normal `write()`: `write()` allocates PMEM pages, initializes them, and appends a write log entry to add the new pages to the file. A `patch()` uses the pages provided by the application, skips the allocation and initialization, and appends the write entry.

Implementing `alloc_pmem()` and `patch()` this way means the pages used by `patch()` belong to two files – the temporary file used to allocate the buffer and the target file. Similar to `peek()`, the shared pages should be protected from updates occurring either by modifying the buffer with stores or by modifying the target file with `write()`. To support this, NOVA sets the pages as read-only after they are used by `patch()`, and always performs copy-on-write on modification by `write()`. For non-page aligned accesses and patch sizes, it may be required to overwrite an additional page before and/or after the patched pages. This is easily done by appending additional log entries describing these overwrites.

4.3.2 XFS-DAX file system

XFS is a widely-deployed, high-performance journaling file system. It organizes files with variable-sized extents and maintains a B+tree for each inode to map file offsets to those extents. As an additional mode, XFS-DAX allows direct access to the extents in PMEM, bypassing the page cache layer.

Although XFS-DAX, by default, updates data in-place, it also has facilities to support out-of-place data updates as part of its implementation of "reflink", a feature that allows multiple files to share data pages [49, 88, 11]. Reflink allows sharing pages between files and supports copy-on-write updates when the shared pages are modified, the same mechanisms required by SubZero. Therefore, implementing SubZero in XFS-DAX mainly involved enabling the reflink to work with DAX mode in addition to the page table manipulations to mark PMEM pages as read-only.

4.4 Evaluation

We evaluate the performance of SubZero against copy-based `read()` and `write()`, as well as DAX-`mmap()`, on two PMEM file systems, NOVA [112] and XFS-DAX [49] under Linux kernel 4.19. We answer the following questions:

- How much speedup do `peek()` and `patch()` achieve?

- How much effort is required to modify applications to use SubZero?
- How much does SubZero improve performance on real applications?

We performed experiments on a dual-socket machine provided by Intel Corporation. The CPUs are 24-core Cascade Lake engineering samples with a similar spec as the previous-generation Xeon Platform 8160. Each core has exclusive 32 kB L1 instruction and data caches, and 1 MB L2 caches. All cores share a 33 MB L3 cache. Each CPU has two iMCs and six memory channels (three channels per each iMC), and each memory channel is attached with a 32 GB Micron DDR4 DIMM and a 256 GB Intel Optane DC Persistent Memory Module (Optane DCPMM). Overall, the system has 384 GB of DRAM and 3 TB of PMEM. Every experiment is configured to access the DRAM and PMEM in the same socket.

4.4.1 Micro-benchmarks

To understand how SubZero performs compared to other legacy IO operations, we compare the performance of SubZero operations against that of `read()`, `write()`, and DAX-`mmap()`-based IO methods. To calculate the latency, we read or write a large number of files with each IO method while varying the IO size from 4 kB to 4 MB. We repeated this single-thread benchmark 100 times and report the average latency.

Most importantly, understanding the performance benefit of SubZero requires investigating not only the cost of IO system calls themselves, but also that of their related operations – memory allocation, population, consumption, etc. For this, the latency in all IO methods in our experiments includes the time to allocate/free the buffer (if applicable). In case of read, the latency also includes the time to load all bytes from the target file after each IO method. In case of write, the latency also includes the time to persist all bytes to the target file.

Read Latency Figure 4.3 compares the latency of `peek()` against `read()` and DAX-mmap()-based load. Here, we differentiate two different read() cases: read denotes cases where the DRAM buffer is *not* reused for subsequent read() operations whereas read-opt represents cases where the buffer is reused either by the application or the glibc malloc().

As a result, the relative speedup of peek() largely depends on whether the buffer is reused in using read(): peek() outperforms read by 1.6–2× and 1.6–1.7× in NOVA and XFS-DAX, respectively while the speedup of peek() over read-opt reduces to 6–17% in both file systems. The reduction in speedup mainly comes from the operating system overheads avoided by the buffer reuse in read(). Otherwise, any new buffer allocation normally requires a system call (i.e., mmap() or sbrk()) and page fault overheads. Overall, the result indicates that replacing read() with peek() is mostly beneficial when the target program does not reuse the DRAM buffer frequently or it is hard to do so due to the memory allocation pattern in the program.

Compared to mmap(), peek() shows comparable performance (-5–10%) since the underlying page table mapping mechanism is fundamentally identical.

Write Latency Figure 4.4 compares the latency of patch() against write() and DAX-mmap()-based store. As with read, write-opt denotes the buffer reuse case. For patch(), -a and -ua indicate page-aligned and unaligned access, respectively. After write() and patch(), we call fdatasync() to ensure the written data is made durable.

Of note, aligned patch() (patch-a) outperforms write() up to 2.8× and 2.2× in NOVA and XFS-DAX, respectively. The speedup is small when the access size is small, but it starts increasing as the access size increases. When the buffer is reused during write(), aligned patch() performs 10–30% slower for 4 kB, but starts outperforming from 16 kB, and achieves up to 1.7× in both file systems. Unaligned patch() (patch-ua) performs slower than aligned patch() due to the additional

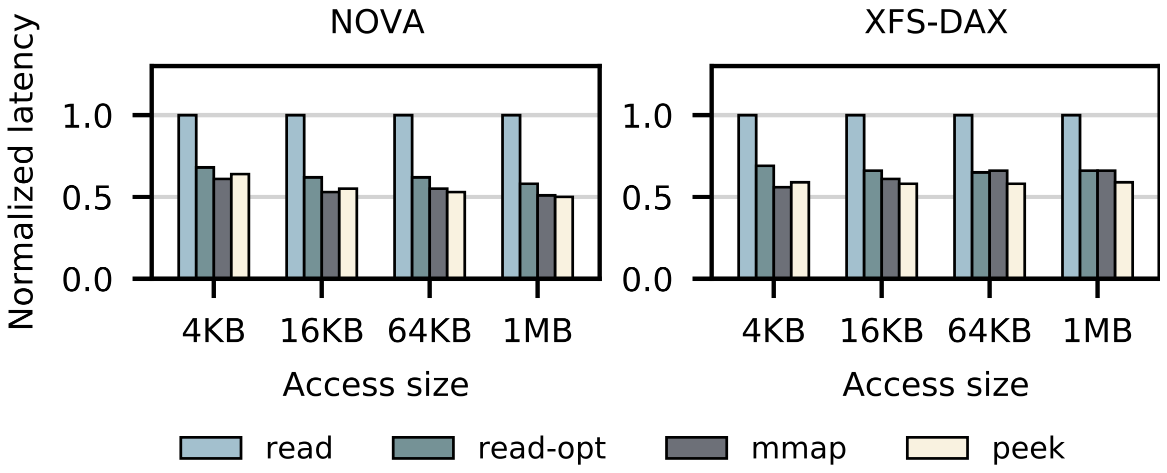


Figure 4.3. Read operation latency. All latencies are normalized to `read` latency. Lower is better.

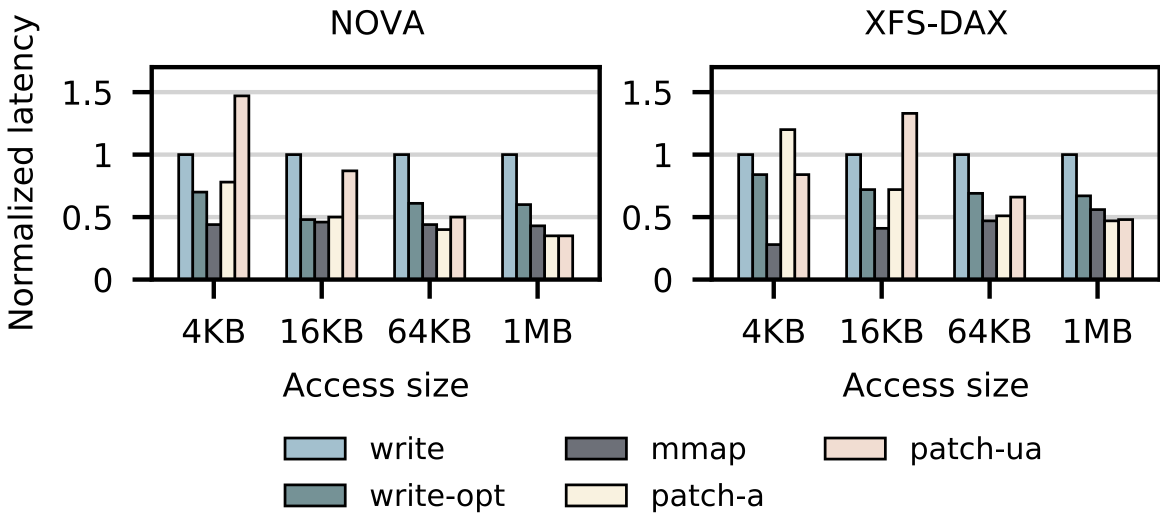


Figure 4.4. Write operation latency. All latencies are normalized to `write` latency. Lower is better.

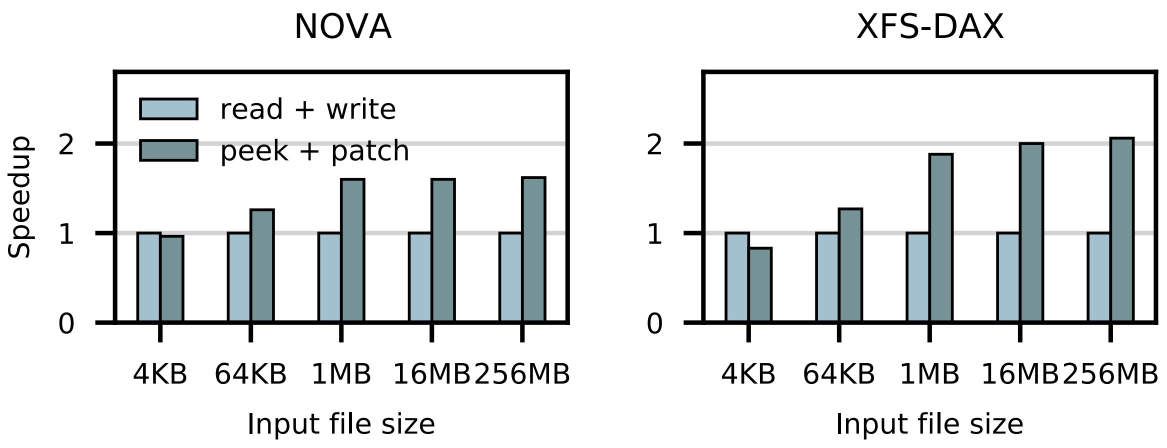


Figure 4.5. Boosted computation between PMEM files. Combining `peek()` and `patch()` allows computation between PMEM files without any copies, therefore boosts the performance.

overheads from copying head and tail pages. The gap reduces to close to zero as the access size grows. 4 kB unaligned `patch()` falls back to the copy-based `write()` operation since there is no page to share.

Compared to `mmap()`, aligned `patch()` underperforms 1.8× and 4.3× for 4 kB in NOVA and XFS-DAX, respectively, but the gap reduces as the access size grows. Beyond 64 kB, aligned `patch()` outperforms `mmap()`. The main reason is that `patch()` saves the page fault cost by using pre-faulted buffer pages whereas `mmap()` includes the cost in the critical path. Also, despite its higher speed on the smaller accesses, `mmap()` only provides the atomicity for 8 bytes while `patch()` provides the crash-consistency of each IO operation.

Uudecoding files We evaluate the example code seen in Figure 4.2 where a combined use of `peek()` and `patch()` can boost the performance of computation between different PMEM files. We used the same code in Figure 4.2 with base64 encoding schemes with results in Figure 4.5. As a result, using `peek()` and `patch()` over `read()` and `write()` achieved up to 1.6× and 2× speedups on NOVA and XFS-DAX, respectively.

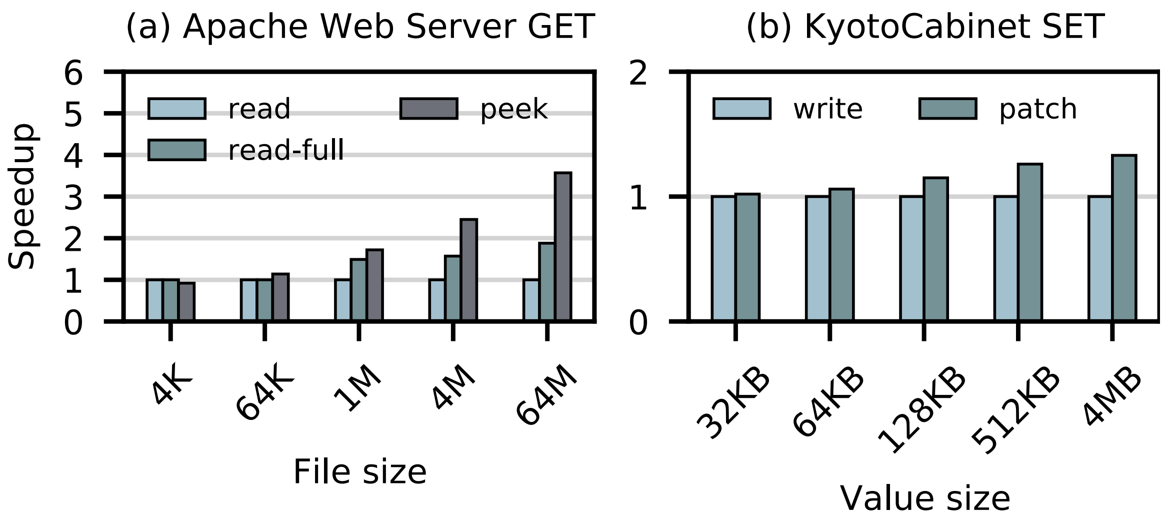


Figure 4.6. Application performance. SUBZERO boosts application performance by a wide margin with small code changes.

4.4.2 Applications

We explored the impact of SUBZERO on real applications with two examples, Apache Web Server and Kyoto Cabinet.

Apache Web Server To apply SUBZERO to Apache Web Server, we modified the Apache Portable Runtime library, a supporting library for the web server, to use `peek()`. We measured the performance of HTTP GET request serving static files using a built-in web performance benchmark, ApacheBench [1], on our modified NOVA. When reading file contents with `read()`, Apache Web Server uses an 8 kB buffer by default. For `peek()`, we set the access size to be the same as the file size since it performs the best. Figure 4.6 (a) plots the throughput of GET requests normalized to the `read()`-based method. As a result, `peek()` outperforms `read()` from 1 MB (1.7×) and achieves up to 3.6× better throughput. Since the default `read()` suffers the overhead of frequent system calls, we increased the `read()`'s buffer size to be the same as the file size (`read-full`). On this setting, `peek()` still offers up to 1.9× speedup.

To benefit from these speedups, `peek()` required 48 LOC changes.

Kyoto Cabinet Kyoto Cabinet [40] (KC) is a high performance database library that stores variable-sized key-value records in a single file. KC supports fast access to the records via hash table or B+tree, and makes every operation transactional using write-ahead logging (WAL) to a separate log file. By default, KC updates the database and the WAL file using the `write()` operation. To demonstrate the benefit of SUBZERO, we applied `patch()` to the hash table-based HashDB database to speed up updating key-value records in the underlying database file and measured the throughput of transactional SET operations. Note that our modified KC performs unaligned patches to the database file as each record is padded with preceding metadata fields. In Figure 4.6 (b), `patch()` performs similar to the `write()`-based operation for value sizes 32, 64 kB, but after the 128 kB value, it begins to outperform `write()` and the speedup monotonically increases up to 1.3×.

To experience these speedups with SUBZERO, we required 57 LOC changes.

4.5 Related Work

Avoiding data movement in storage systems The techniques to avoid data movement have been explored in a variety of systems, ranging from persistent storage to DRAM. For file systems, Ext4 supports an `ioctl` operation, `EXT4_IOC_MOVE_EXT`, to allow swapping extents between files by modifying inodes [7, 2]. A recent system SplitFS [53] extended this feature in Ext4-DAX but still requires the data movement that SUBZERO avoids since it supports copy-based `read()`, `write()` semantics. Other file systems [49, 88, 11] have similar functionality called "reflink" that could be a useful facility to implement SUBZERO. The technique to remap pages to avoid data movements has also been explored for flash drives [110, 81, 54]. For DRAM, operating systems, such as OSX, that inherit the Mach [8] support memory copy via `vm_copy` that remaps the regions as copy-on-write pages [4].

PMEM allocator For fast and efficient PMEM allocation, several schemes have been proposed from both industry and academia. PMDK [85] is an open-source, PMEM library bundle from Intel. It offers large virtual address pools by memory-mapping to PMEM files. Schewalb *et al.* [91] proposed a general-purpose memory allocator for PMEM that combines both DRAM and PMEM for fast allocation and recovery. Makalu [16] offers an integrated allocator and garbage collector that avoids memory leaks on failures while offering better integration with existing PMEM libraries [24]. Pallocator [82] improves defragmentation by maintaining multiple PMEM regions instead of having a single large pool. Compared to these allocators, our allocator focuses on highly fast allocations (by pre-faulting pages and partitioning) and the correct recovery of both buffer and regular files. The SUBZERO allocator is currently very simple; it could be improved by incorporating techniques from these projects.

4.6 Summary

We have described and implemented SUBZERO, a new IO mechanism that avoids most or all data movement for reads and writes to PMEM-backed files. In addition to minimizing movement, our implementation of SUBZERO provides both fast read access and strongly consistent updates. Our evaluation shows that SUBZERO outperforms copy-based `read()` and `write()` by a wide margin. In summary, SUBZERO IO is a straight-forward way for programmers to improve their applications' performance on PMEM file systems.

Acknowledgements

This chapter contains material from "SubZero: Zero-Copy IO for Persistent Main Memory File Systems", by Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and

Steven Swanson, which appeared in the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2020). The dissertation author is the primary investigator and the first author of this paper.

Chapter 5

Blaze: Fast Graph Processing on Fast SSDs

Out-of-core graph processing enables the processing of large graphs that do not fit in the available main memory of a single machine by judiciously moving data between memory and storage. The design of out-of-core graph processing systems has evolved for nearly a decade [64, 89, 44, 123, 121, 67, 52] with a strong focus on optimizing IO performance to minimize the overhead of slow storage access. With significant improvements in storage technology, the design of these systems has also been tailored to benefit from the improved performance of more advanced devices. Well-optimized, out-of-core graph processing systems have shown that they provide attractive performance with lower cost and complexity compared to the complex distributed graph processing solutions that spread the graph in the memories of multiple machines [43, 56, 69, 118].

The design of out-of-core graph processing systems now faces new challenges and opportunities as more performant storage technologies emerge. For example, modern SSDs like Intel Optane SSD or Samsung's Z-NAND offer an order-of-magnitude higher bandwidth compared to conventional SSDs. The most critical aspect of these new devices is their improved bandwidth and their symmetric performance between sequential and random IO. We refer to these modern SSDs as Fast NVMe Drives (FNDs), the same terminology used by previous literature [61].

In this work, we present Blaze, an open-source, out-of-core graph processing system optimized for FNDs.[1] Specifically, Blaze aims to keep the FNDs constantly saturated to achieve high performance. For this, Blaze introduces a novel scatter-gather scheme called *online binning* that propagates values among graph vertices without the synchronization overhead while achieving good load balance, the goal previous techniques like synchronization and message passing cannot achieve simultaneously. For balanced IO, Blaze uses page-interleaved Compressed-Sparse Row (CSR) format which helps increase IO utilization from multiple SSDs while minimizing IO amplification. Finally, Blaze allows the programming of efficient out-of-core graph algorithms under the well-known API, EDGEMAP and VERTEXMAP, first introduced in Ligra [96]. We extend them to be efficiently used in out-of-core graph processing.

We evaluate Blaze against two state-of-the-art, open-sourced, out-of-core graph processing systems, FlashGraph [121] and Graphene [67]. These systems are also designed for random IO unlike early-generation out-of-core graph processing systems [64, 89, 123]. Compared to FlashGraph and Graphene, Blaze offers substantial speedups (up to 13.6×) in a wide variety of workloads as we describe in Section 5.4.

Overall, we make the following contributions in this paper.

- An analysis of two recent out-of-core systems, FlashGraph [121] and Graphene [67], revealing their performance problems on FNDs
- A novel atomic-free, scatter-gather scheme called *online binning* that applies graph algorithms on disk-resident graphs with low CPU overhead and high load balance
- An extension of EDGEMAP API to the out-of-core graph processing stack
- An open-sourced implementation of Blaze

[1]The code is available at https://github.com/NVSL/blaze.

This paper is organized as follows. Section 6.1 describes the background and motivation of this work. Section 5.2 discusses the root cause of the low performance of existing systems on FNDs. Section 5.3 describes the design and implementation of Blaze. Section 5.4 describes the experimental setup and results. Section 5.5 discusses related works and Section 6.4 concludes.

5.1 Background and Motivation

In this section, we describe the background on out-of-core graph processing and the performance characteristics of modern FNDs. Then we further discuss the performance problems of current out-of-core graph processing systems when they run on FNDs, motivating our work.

5.1.1 Out-of-core Graph Processing

Out-of-core graph processing enables the processing of very large graphs that do not fit in the main memory of a single machine – by placing the graph on the secondary storage and conducting frequent IO to process graph algorithms on storage-resident graphs. Compared to the distributed graph processing that places graphs on an aggregated memory of multiple machines, out-of-core processing offers similar or better performance without the need to deal with the complexity of distributed computing [64, 89, 123, 121, 67, 72].

Current systems The design of out-of-core graph processing systems have evolved in step with advances in storage performance. For instance, out-of-core systems designed around the early 2010s were optimized for sequential disks [64, 89]. To maximize the IO performance, these systems access disk-resident graphs *sequentially* at the cost of accessing more data than necessary. Even with this potential access amplification, they benefit from sequential access due to the significant performance gap between sequential and random access on early-generation disks.

Table 5.1. Target graphs. The number of vertices (|V|) and edges (|E|) is in millions. "Short" denotes short names for datasets.

Dataset	Short	|V|	|E|	Distribution	Diameter	Type
rmat27	r2	134	2147	power	10	synthetic
rmat30	r3	1074	17180	power	11	synthetic
uran27	ur	134	2147	uniform	10	synthetic
twitter	tw	61	1468	power	75	real
sk2005	sk	51	1949	power	205	real
friendster	fr	124	1806	power	56	real
hyperlink14	hy	1727	64422	power	790	real

However, more recent systems [121, 67] make a different tradeoff since storage devices started offering fast random access with little performance gap with sequential access. These systems do not place a high priority on issuing large sequential IO, achieving lower IO amplification than prior systems. Blaze follows the same principle but further optimizes for FNDs to maximize the benefit of fast random IO that existing systems fail to leverage.

Processing models Out-of-core graph processing systems are classified into two models based on where they keep the vertex data. The first model is the fully-external model where the vertex data is kept on storage along with the edges. The second model is the semi-external model where the vertex data is kept fully in DRAM.

The choice between two models is determined by the available memory budget for a machine and the size of target graphs. Performance-wise, the systems with the semi-external model often outperform the ones with fully-external model as less IO is required for the former.

Blaze adopts the semi-external model for better performance while minimizing the use of DRAM.

5.1.2 Target Datasets

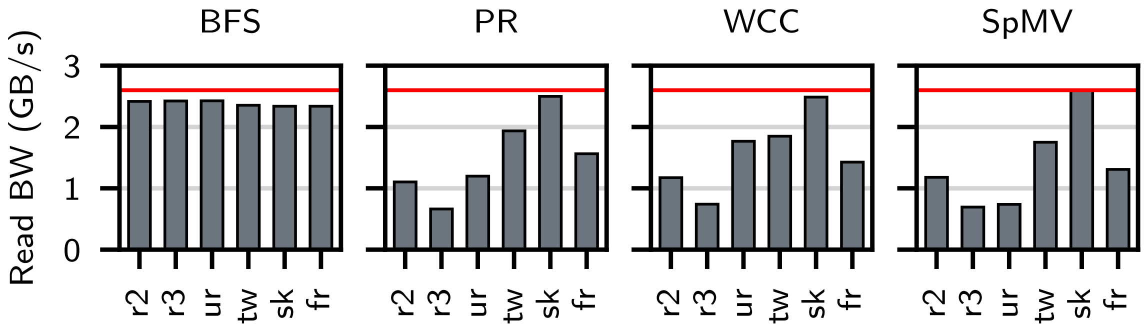
Table 5.1 shows our target datasets throughout this paper. We chose these input graphs as they are topologically diverse and different in size. The *rmat27*, *rmat30*, and *uran27* graphs are synthetic while *twitter*, *sk2005*, *friendster*, *hyperlink2014* are from real-world. Six graphs except uran27 follow a power-law degree distribution while uran27 follows a normal degree distribution. The uran27 is the most adversarial graph [14] as it has *no locality* – there are no popular vertices (no temporal locality) and neighbors are not close to each other (no spatial locality), so it well represents the other extreme in our choice of input graphs.

5.1.3 Issues with Current Out-of-core Systems

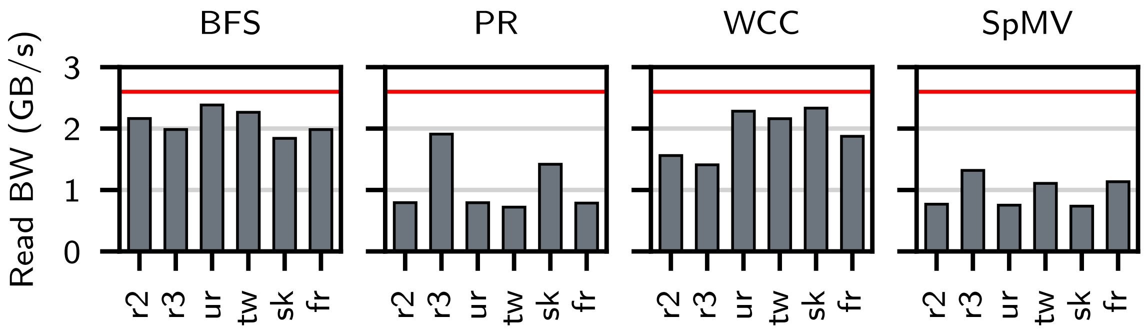
Current out-of-core graph systems cannot efficiently utilize the FND's bandwidth. FlashGraph [121] and Graphene [67], two recent out-of-core graph processing systems optimized for random IO, illustrate this problem – they fail to utilize the high throughput of FNDs, leading to a suboptimal performance on representative workloads.

We confirm this by measuring the average IO bandwidth utilization of both systems on an Intel Optane SSD with various graph workloads (Figure 5.1). We used six graph inputs (r2, r3, ur, tw, sk, fr) from Table 5.1 and ran five queries – Breadth-First Search (BFS), PageRank (PR), Weakly-Connected Components (WCC), Sparse Matrix Vector Multiplication (SpMV), and Betweenness Centrality (BC). For all measurements, we used 16 threads for a fair comparison.

In both FlashGraph and Graphene, IO utilization significantly varies by input graph and query. Both systems achieve high IO bandwidth regardless of the input graph for BFS. However, for PR, WCC, SpMV – more complex queries than BFS – both systems show low IO bandwidth depending on the underlying graph. In the worst case, FlashGraph achieves only 23% of the device bandwidth for PageRank on rmat30 graph



(a) FlashGraph



(b) Graphene

Figure 5.1. Underutilized IO in FlashGraph and Graphene. Red line: the maximum read bandwidth of Optane SSD.

Table 5.2. System comparison. Blaze avoids the root causes of low IO utilization on FNDs.

Systems	Skewed computation	Skewed IO	Fast IO & slow computation
FlashGraph [121]	Yes	No	No
Graphene [67]	No	Yes	Yes
Blaze	No	No	No

while Graphene achieves 30% of it for PageRank and SpMV on various graphs.

In the following section, we investigate the root cause of why FlashGraph and Graphene suffer such low IO utilization on FNDs.

5.2 Reasons of Low IO Utilization in Current Systems

The root cause of low IO utilization in FlashGraph and Graphene on FNDs are *skewed computation*, *skewed IO*, and *fast IO, slow computation*. We elaborate on each case in more detail.

5.2.1 Skewed Computation

Parallel graph processing requires synchronization to avoid data races when updating the algorithm-specific vertex data concurrently with multiple threads [78, 96]. However, synchronization primitives like `compare-and-swap` incur high CPU overhead, which potentially leads to low IO utilization of FNDs in out-of-core processing. A well-known alternative that does not require synchronization for each update is message passing technique. FlashGraph [121] adopts message passing by assigning a message queue to each vertex, and assigning each vertex to one of the computation threads based on the vertex ID. FlashGraph processes these messages at the end of each iteration to update the algorithm-specific vertex data and generate a set of vertices which will be activated in the next iteration.

The problem with the message passing scheme in FlashGraph is the potential risk of *skewed computation* on power-law graphs – some threads need to process more messages than others because certain vertices have much higher number of neighbors than other vertices on these graphs. In out-of-core graph processing, all activities including IO must wait until the straggler thread finishes the processing of messages in each iteration. Crucially, FND can potentially finish all IO requests faster than the straggler thread so it may frequently remain idle over iterations.

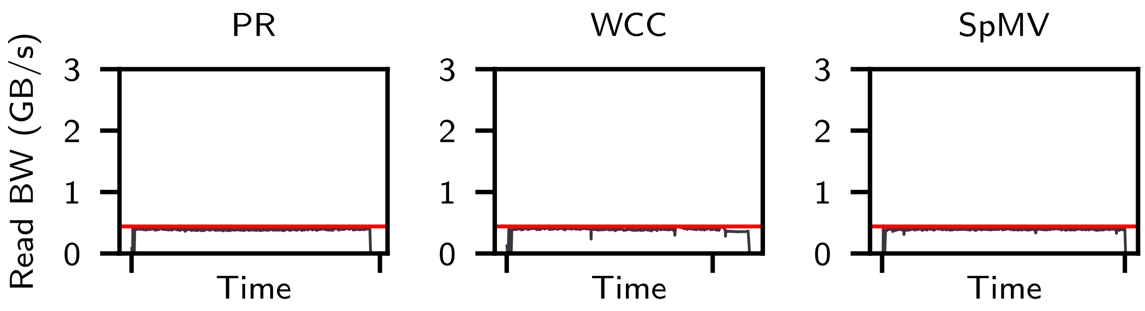
We observe this phenomenon on Optane SSD as shown in Figure 5.2. On NAND SSD (Figure 5.2 (a)), FlashGraph fully utilizes the device's read bandwidth on three queries, PR, WCC, and SpMV. However, for the same queries, it fails to issue any IO to the Optane SSD at the end of each iteration (the period where the read bandwidth remains zero) due to the straggler thread still processing a large volume of messages (Figure 5.2 (b)).

Mitigating this problem requires balancing the workload – the messages passed among vertices – between threads but achieving this without synchronization is not straightforward. We solve this problem in Blaze with a synchronization-free, online binning technique we describe in Section 5.3.1.

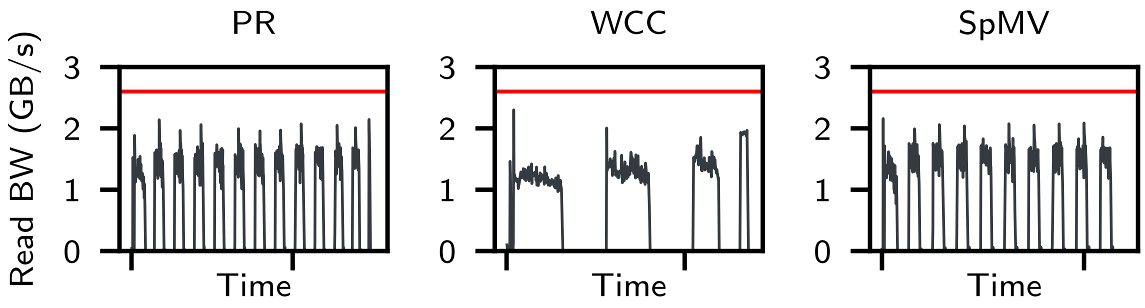
5.2.2 Skewed IO

Another problem that leads to low IO utilization is skewed IO. We observe this problem in Graphene.

Skewed load of IO across multiple disks is another source of low IO utilization. In synchronous graph processing where edges are distributed in multiple disks, the maximum aggregate IO bandwidth is achieved when all disks are kept busy at all times. When IO is not balanced, however, it leaves some disks to wait until other disks complete their requests. We find that Graphene suffers this *skewed IO* problem due to its topology-aware partitioning scheme.



(a) FlashGraph on NAND SSD



(b) FlashGraph on Optane SSD

Figure 5.2. Idle IO periods in FlashGraph on Optane SSD. Red line: the maximum read bandwidth of Optane SSD. Input graph: rmat30.

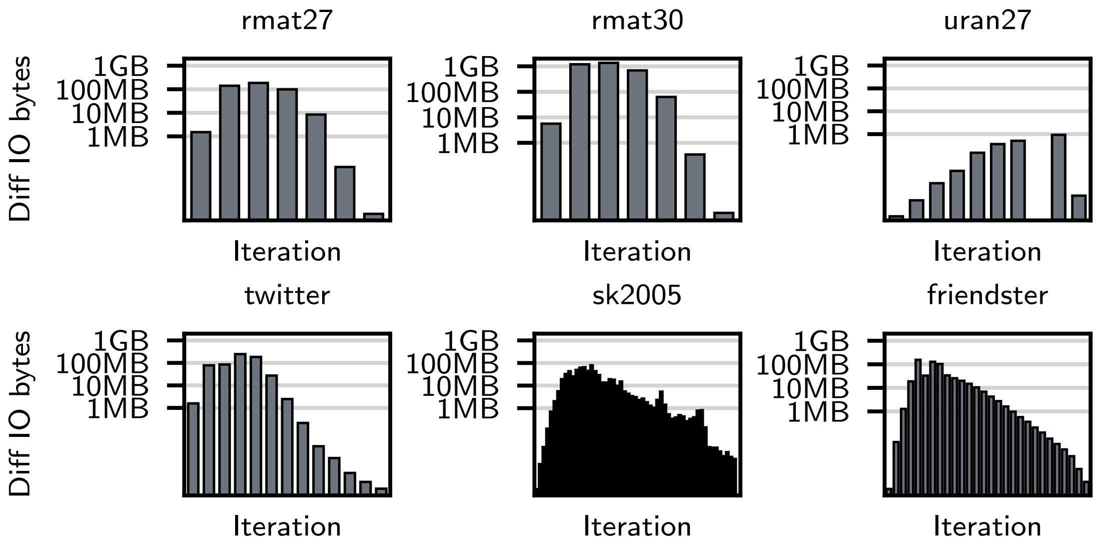


Figure 5.3. Skewed IO in Graphene. y-axis: *max* − *min* IO bytes between eight SSDs for each iteration.

Graphene adopts 2-D partitioning of a graph with the goal of producing partitions with the same number of edges. Then it distributes these partitions on multiple disks in a way that each disk has the same number of partitions, making each disk have an equal number of edges.

Despite having a balanced partition distribution, Graphene suffers highly skewed IO on algorithms that employ *selective scheduling* of edges. Selective scheduling means that only a subset of the total edges are traversed in a given iteration, a common technique to increase algorithm efficiency by only accessing the necessary edges for a given algorithm goal. In Graphene, these algorithms end up accessing edges on certain disks more than those on others, leading to skewed IO.

Figure 5.3 shows the skewed IO of Graphene on BFS that employs selective scheduling over iterations. On the y-axis, the figures show the maximum difference between 8 disks in terms of the IO bytes each disk must process in a given iteration. For example, a bar with 10 MB of height means that the disk with the largest amount of IO tasks has to do 10 MB of more IO than the disk with the smallest amount of IO, so the higher bar means IO is more skewed. We observe that Graphene suffers the skewed IO on all power-law graphs. On the uran27 graph with uniform degree distribution, the difference in IO is less than 1 MB. However, on other graphs that follow power-law, the maximum IO difference goes up to more than 100 MB. The impact is more dramatic when considering the ratio, not just the absolute bytes – A disk has to conduct up to 1.7–2.1× (depending on the input graph except uran27) more IO than another disk.

Based on these results, we conclude that the topology-aware graph partitioning adopted by Graphene incurs the skewed IO problem when running algorithms with selective scheduling. We mitigate this problem with topology-agnostic graph partitioning based on the page interleaving as we describe in Section 5.3.5.

5.2.3 Fast IO, Slow Computation

Graphene's low IO utilization also stems from its thread assignment policy which leads to the *fast producer and slow consumer* problem. Graphene equally devides cores across IO and computation – a pair of cores are assigned to each SSD, one for IO and one for computation. For slow SSDs, this scheme still helps maximize the IO bandwidth by assigning a dedicated thread for each SSD.

However, two threads strictly assigned for each SSD are not sufficient for FNDs because they cannot saturate the bandwidth of FNDs. When the producer (IO thread) sends IO buffers filled with on-disk pages faster than the consumer (computation thread) can process, the free IO buffers soon become unavailable, which in turn blocks the IO thread from issuing more IO requests.

We measure the impact of this problem by comparing the speed of various single-threaded graph computations with the read bandwidth of various storage devices (Figure 5.4). Compared to slow storage like NAND SSD, single-threaded graph computation is fast enough on a set of workloads. However, it does not keep up with

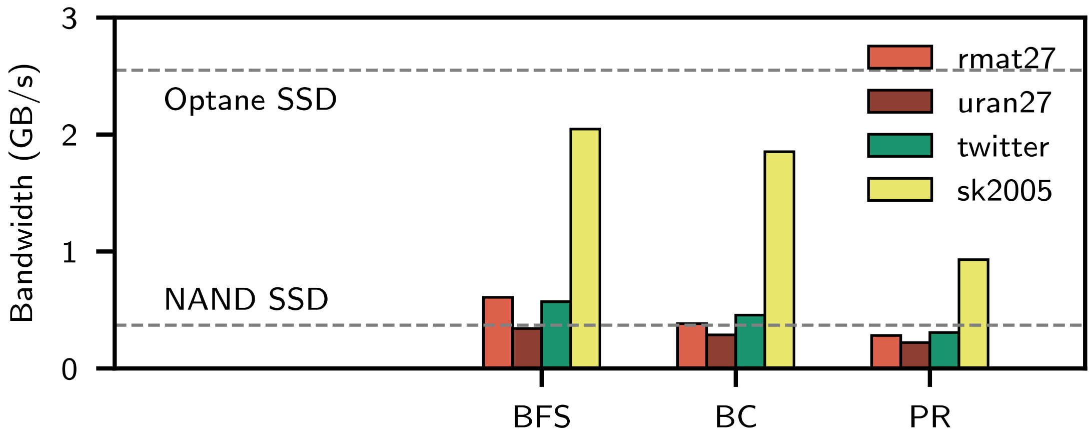


Figure 5.4. Single-threaded graph computation speed (bars) vs. IO bandwidth (lines).

the speed of Optane SSD on all workloads we measured. The result implies that enough threads must be assigned for computation to constantly saturate the underlying FND in out-of-core graph processing.

5.3 Blaze Framework

Blaze supports high-performance graph analytics on FNDs by constantly saturating the underlying IO bandwidth, a challenge that was not achieved by current systems. The key to achieving this goal is the low overhead, scatter-gather scheme called *online binning* that processes user-provided graph computations without synchronization while achieving load balance among threads. In addition, Blaze achieves balanced IO among multiple SSDs by partitioning the input graph with page interleaving (RAID 0) that balances IO well on a variety of workloads. Blaze abstracts these mechanisms in the well-known, in-core graph processing API, EDGEMAP and VERTEXMAP [96], to enable the programming of efficient out-of-core graph algorithms without the need to handle complex IO executions.

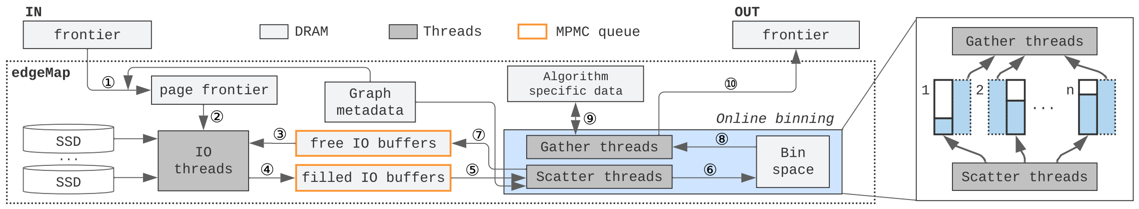


Figure 5.5. Out-of-core EDGEMAP engine in Blaze.

5.3.1 Online Binning

An IO-efficient execution of EDGEMAP relies on low-overhead graph computation which we enable with the technique we call *online binning*. The idea of using bin data structure in graph processing is inspired by propagation blocking [13] but we adapt this idea to out-of-core graph processing.

A bin is a struct kept in DRAM that holds multiple bin records where each bin record is a $\langle vertex\_id, value \rangle$ pair. During execution, Blaze creates a bin record for each algorithm-specific scatter function with the destination vertex id and the value returned by the scatter function. Then Blaze appends the record to the corresponding bin ($bin\_id = vertex\_id \bmod bin\_count$). Once a bin becomes full, Blaze pushes it to a concurrent queue called `full_bins` to allow gather threads to process the records in the full bins. Each gather thread processes one full bin in its entirety. Most importantly, Blaze ensures that *no two gather threads process the same bin at the same time* and this avoids the need to synchronize between gather threads.

To maximize the performance of online binning, Blaze adopts several optimization techniques. First, Blaze uses a small fixed size, per-CPU buffer [13] to reduce the synchronization overhead while binning, where the buffer allocates memory space for each bin. Blaze first appends the bin record to this buffer, and once it becomes full, Blaze copies the records in the buffer to the corresponding bin in batch. Second, Blaze uses MPMC queue for `full_bins` for highly concurrent push/pull of full bins between scatter and gather threads. Third, Blaze implements each bin as a pair of bins to ensure the forward progress of both scatter and gather threads. Once one of the pair becomes full, its pointer is appended to the `full_bins` queue for gather threads while the other bin serves scatter threads. A scatter thread is blocked until a gather thread finishes the processing of the full bin and returns it to the empty state.

Online binning has several configuration parameters including bin count, bin size, and the ratio between scatter and gather threads. Data in Section 5.4.5 shows that performance is robust across a wide range of values, so precise tuning is not required. In particular, our data show that *one thousand bins*, *0.05$\times$ of the input graph size for bin space*, and *an equal number of scatter and gather threads* will provide good performance in general and that more careful tuning improves performance by, at most, 5%.

5.3.2 Programming API

To support the programming of efficient out-of-core graph algorithms, Blaze provides two key APIs, EDGEMAP and VERTEXMAP. The APIs were first introduced by Ligra [96] in-core graph processing framework and have shown that they can be used to express a broad range of efficient, parallel graph algorithms [96, 95, 33] for in-memory graph processing. We extend them to enable efficient out-of-core graph processing while hiding the binning-based execution entirely from the user.

$$\textsc{edgeMap}(graph : Graph,$$
$$frontier : VertexSubset,$$
$$f_s : (vertex \times vertex) \rightarrow value\_type,$$
$$f_g : (vertex \times value\_type) \rightarrow bool,$$
$$cond : vertex \rightarrow bool,$$
$$output : bool) : VertexSubset$$

Executes two edge functions, f_s and f_g, to the edges whose source vertices are in the given $frontier$. Users provide the scatter function f_s that returns an algorithm-specific value to scatter it to neighboring vertices. The value is scattered to the gather threads only when *cond* returns true with the destination vertex ID as argument. Users also provide the gather function f_g that accumulates the scattered values to the algorithm-specific data array. When the $output$ is true, EDGEMAP creates an output frontier and pushes the destination vertex ID to the frontier if f_g returns true.

The scatter and gather functions communicate intermediate data via bin data structure provided by the online binning mechanism. This ensures that scatter and gather steps are executed without synchronization overhead while achieving load balance among worker threads.

$$\textsc{vertexMap}(frontier : VertexSubset,$$
$$f : vertex \rightarrow bool) : VertexSubset$$

Applies a vertex function f to each vertex in the $frontier$. It conditionally filters out the vertices from the frontier when f returns true, and returns a new frontier. In Blaze, VERTEXMAP executes entirely in memory as all vertex-related data is placed in memory. In most algorithms, VERTEXMAP is used along with EDGEMAP alternatively in an iteration to update vertex values and reduce the next frontier size. In Section 5.3.4, we describe how BFS, PageRank, and WCC algorithms use both EDGEMAP and VERTEXMAP functions together in more detail.

5.3.3 Out-of-core EDGEMAP Execution

Figure 5.5 shows the architecture of Blaze's EDGEMAP engine and how an EDGEMAP function is executed in an out-of-core fashion along with online binning.

With the frontier as input, an EDGEMAP function starts execution by first transforming the given frontier into the $page frontier$, a data structure that contains the disk page IDs that contains the target vertex IDs in the frontier (step 1). Blaze uses all available threads to accelerate this transformation before starting issuing IO requests. Once the page frontier is ready, IO threads start fetching the page IDs from it and send IO requests to the underlying SSDs (step 2) with the free IO buffers (step 3). Blaze uses one thread for each SSD and maintains the page frontier for each SSD. Once the corresponding disk pages are fetched into the buffers (step 4), online binning comes into play – the scatter threads get these fill buffers (step 5), append the records to the corresponding bins (step 6), and return the IO buffers back to the free IO buffer pool (step 7). Concurrently, the gather threads fetch the *full* bins (step 8) and apply the records in the bins into the algorithm-specific, vertex data (step 9). Finally, the gather threads returns a new frontier if required by the caller of EDGEMAP.

To support the fast communication of IO buffers between IO threads and computation threads (scatter, gather threads), Blaze uses a concurrent MPMC (multi-producer, multi-consumer) queue. Blaze maintains two queues, one for free IO buffers, and the other for filled IO buffers, each of which contains the address of the buffer page.

Blaze uses two types of frontier, *VertexSubset* and *PageSubset* for the vertex frontier and page frontier, respectively. Both types abstract the sparse and dense format and switch between them internally depending on the density of the members. Both types use concurrent set data structure when the members are sparse and use bitmap when the members are dense, similarly in Ligra [96]. *PageSubset* is only used internally for IO and not exposed to the users.

For IO execution, Blaze issues IO requests based on the page IDs contained in the page frontier. For continuous pages, Blaze issues only *small contiguous IO* unlike FlashGraph [121] and Graphene [67]. Concretely, Blaze merges up to four contiguous 4 kB pages as larger IO request is not beneficial on FND. Rather, it is studied in Graphene [67] that the large IO significantly increases the Asynchronous IO submission time. Also, Blaze does not attempt to merge non-consecutive pages even if they are within a certain threshold [67] – On FNDs, 4 kB random IO is already fast enough such that there is little incentive to issue large IO requests at the cost of accessing non-target pages.

5.3.4 Examples

BFS Algorithm 1 shows a parallel out-of-core BFS algorithm written in Blaze's API. The user provides two edge functions, SCATTER and GATHER. Leveraging the online binning internally, they cooperatively update the *Parent* array without synchronization overhead. Specifically, SCATTER function examines input edges and returns the source vertex ID. To reduce unnecessary propagation of values, the user also provides COND function – SCATTER is executed only when the destination vertex has not been visited yet by checking if COND returns true. Then GATHER function receives the value (v) along with the associated destination ID (d). If the destination vertex has not been visited ($Parent[d] == -1$), GATHER updates the parent array with the source vertex ID and returns 1, activating the current destination vertex in the next iteration. Finally, these functions are used in the main BFS function as arguments to the EDGEMAP that

Algorithm 1. Breadth-First Search

```
$Parent = \{-1, ..., -1\}$                                      ▷ initialize all to -1's

procedure SCATTER($s$, $d$)                                     ▷ scatter function
    return $s$
end procedure

procedure GATHER($d$, $v$)                                      ▷ gather function
    if $Parent[d] == -1$ then
        $Parent[d] = v$
        return 1
    end if
    return 0
end procedure

procedure COND($d$)                                             ▷ conditional function
    return $Parent[d] == -1$
end procedure

procedure BFS(G, s)                                             ▷ s is the root
    $Parent[s] = s$
    $F = \{s\}$
    while $\neg F.empty()$ do
        $F = $ EDGEMAP($G, F,$ SCATTER, GATHER, COND, $true$)
    end while
end procedure
```

iteratively runs until the frontier F becomes empty.

PageRank Algorithm 2 shows an example of PageRank that implements PageRank-delta algorithm [96, 69], a variant of PageRank in which vertices are active in an iteration only if they have accumulated enough change in their page rank values. In our implementation, EDGEMAP propagates the delta value of each vertex, normalized with its out-degree, to the out-going neighbors in SCATTER and accumulates those values in GATHER without synchronization. Then VERTEXMAP applies the accumulated delta values kept in *ngh_sum* to the *delta* array and filters out vertices whose change in the page rank value in p is less than a given threshold e, as implemented in APPLYFILTER. The EDGEMAP and VERTEXMAP alternately run until no vertex is active in the frontier.

WCC Algorithm 3 shows the shortcutting label propagation algorithm running on an undirected graph [99] implemented in Blaze API. While SCATTER and GATHER updates the destination vertex value with the smaller vertex ID (normally as in original label propagation), the shortcutting mechanism in APPLYFILTER conducts pointer jumping to accelerate the label propagation. In addition, it activates only the vertices that suffered the value change from the previous iteration. With Blaze API, our WCC implementation executes EDGEMAP for both CSR (*outG*) and a transpose of it (*inG*) to propagate vertex values on an undirected graph. The algorithm finishes when no further propagation is required.

5.3.5 Balanced IO

In addition to the low-overhead, balanced computation powered by online binning, Blaze also achieves balanced IO with *page-interleaved, Compressed Sparse Row (CSR) format* – Blaze stripes a CSR graph into multiple SSDs in 4 kB granularity. Page interleaving (RAID 0) is a well-known technique used in various HPC domains to maximize the aggregate bandwidth of underlying devices. We find that it is also effective in out-of-core graph analytics. We reject other topology-aware partitioning

Algorithm 2. PageRank

```
1: $G \leftarrow Input\ graph$
2: $p = \{0,...,0\}$
3: $ngh\_sum = \{0,...,0\}$
4: $delta = \{\frac{1}{V},...,\frac{1}{V}\}$
5: $D \leftarrow 0.85, e \leftarrow threshold$
6:
7: procedure SCATTER($s$, $d$)                                  ▷ scatter function
8:     return $delta[s]/G.get\_degree(s)$
9: end procedure
10:
11: procedure GATHER($d$, $v$)                                  ▷ gather function
12:     $ngh\_sum[d] += v$
13:     return 1
14: end procedure
15:
16: procedure COND($d$)                                         ▷ conditional function
17:     return 1
18: end procedure
19:
20: procedure APPLYFILTER($i$)                                  ▷ vertex function
21:     $delta[i] = ngh\_sum[i] * D$
22:     $ngh\_sum[i] = 0$
23:     if $|delta[i]| > e * p[i]$ then
24:         $p[i] += delta[i]$
25:         return 1
26:     else
27:         return 0
28:     end if
29: end procedure
30:
31: procedure PAGERANK(G)
32:     $F = \{1,...,1\}$                                       ▷ activate all vertices
33:     while $\neg F.empty()$ do
34:         EDGEMAP($G, F$, SCATTER, GATHER, COND, $false$)
35:         $F$ = VERTEXMAP($F$, APPLYFILTER)
36:     end while
37: end procedure
```

Algorithm 3. WCC

```
$Ids = \{0, ..., V-1\}$                                   ▷ initialize all to node ids
$PrevIds = \{0, ..., V-1\}$                               ▷ initialize all to node ids

procedure SCATTER($s$, $d$)                               ▷ scatter function
    return $Ids[s]$
end procedure

procedure GATHER($d$, $v$)                                ▷ gather function
    $orig\_id = Ids[d]$
    if $v < orig\_id$ then
        $Ids[d] = v$
    end if
    return 1
end procedure

procedure COND($d$)                                       ▷ conditional function
    return 1
end procedure

procedure APPLYFILTER($i$)                                ▷ vertex function
    $id = Ids[Ids[i]]$
    if $Ids[i] != id$ then
        $Ids[i] = id$
    end if
    if $PrevIds[i] != Ids[i]$ then
        $PrevIds[i] = Ids[i]$
        return 1
    else
        return 0
    end if
end procedure

procedure WCC(outG, inG)
    $F = \{1, ..., 1\}$                                   ▷ activate all vertices
    while $\neg F.empty()$ do
        EDGEMAP($outG, F$, SCATTER, GATHER, COND, $false$)
        EDGEMAP($inG, F$, SCATTER, GATHER, COND, $false$)
        $F$ = VERTEXMAP($F$, APPLYFILTER)
    end while
end procedure
```

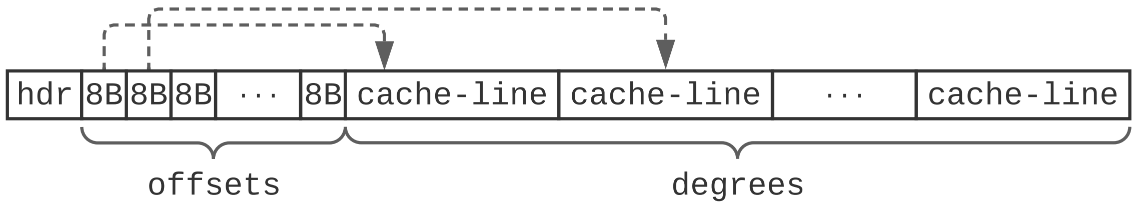


Figure 5.6. Indirection-based graph index in Blaze.

schemes such as 2-D partitioning used in Graphene [67] as they incur imbalanced load across multiple disks when only a subset of edges need to be accessed in each iteration.

5.3.6 Memory Usage

Blaze requires the memory space as follows except the algorithm-specific data.

System-level Regardless of any given workload, Blaze requires a static memory space to allocate IO buffers from. In Blaze, large memory space is not required for IO buffers as scatter threads return the IO buffers quickly enough for IO threads to re-use those buffers. Accordingly, we set the memory space relatively small (64 MB in all workloads) compared to the input graph size we tested.

In addition, Blaze requires a memory space to maintain bins for online binning. We experimentally decide the proper bin size based on our study in Section 5.4.5.

Graph metadata For a given input graph, Blaze maintains an index array and a key-value map in memory for efficient graph access. To keep the graph index array compact, Blaze uses indirection as in Figure 5.6 – Blaze groups sixteen 4 bytes-sized degrees into a single cache-line in the *degrees* region and only keeps the location of the cache-lines in the *offsets* region. With indirection, the offset is retrieved by first looking up the offsets region with *vertex_id* / 16 as key then adding degrees up to *vertex_id* % 16 in the corresponding cache-line in degrees region. This indirection-based index array in Blaze requires about $4\ bytes \times |V|$ of memory.

In addition to the index, Blaze keeps an additional map *page-to-vertex map* to accelerate access to vertex data given a page number. The map returns a $\langle begin\_vertex\_id, end\_vertex\_id \rangle$ pair given an on-disk page number as key. The size of this structure is small as it only requires 8 bytes for each disk page.

5.4 Evaluation

We evaluate Blaze with a variety of workloads, comparing it against two state-of-the-art, open-sourced out-of-core graph processing systems, FlashGraph [121] and Graphene [67]. In addition, we study how Blaze scales with more hardware resources, and how it performs with different binning configurations.

5.4.1 Experimental Setting

Target queries We use the following graph algorithms to evaluate Blaze.

- Breadth-First Search (BFS)
- PageRank (PR) using the delta variant algorithm [69].
- Weakly Connected Components (WCC) using Label propagation [122].
- Sparse Matrix-Vector Multiplication (SpMV)
- Betweenness Centrality (BC) using Brandes's algorithm [21].

We implement these queries based on the implementations in Ligra [96] as both systems share the same API. The difference is that Blaze algorithms require the scatter and gather functions as input to the EDGEMAP function while Ligra requires providing only a single, synchronization-based edge function to the EDGEMAP. Also, Ligra's EDGEMAP is executed purely in memory while Blaze newly introduces the execution of EDGEMAP in an out-of-core fashion.

System configuration Our testbed is a single socket, Intel Xeon Gold 6230 processor (2.1 GHz) with 20 physical cores (no hyperthreading). The machine is equipped with

96 GB of DRAM; one 1.9 TB Intel NAND SSD (DC S3520) and one 960 GB Intel Optane SSD (DC P4800X).

5.4.2 Comparison with Other Systems

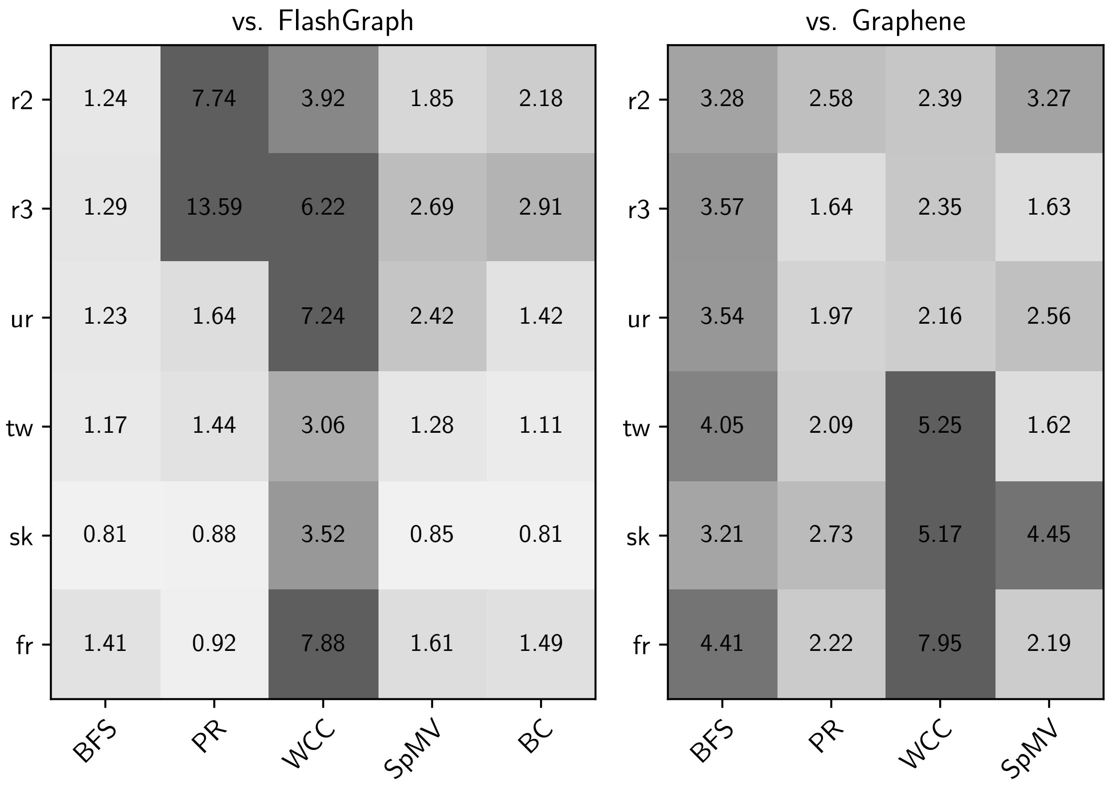
We compare the performance of Blaze with FlashGraph and Graphene on six input graphs stored on an Intel Optane SSD. Among the five target queries – BFS, PR, WCC, SpMV, and BC, we could not compare the result of BC with Graphene since Graphene does not implement BC. For all experiments, we used 16 threads from a single socket for fair comparison.

Figure 5.7 shows the speedup of Blaze over FlashGraph (left) and Graphene (right). Except on sk2005, Blaze generally outperforms FlashGraph, achieving up to 13.6$\times$ speedup when running PageRank on rmat30 graph. On sk2005, Blaze performs 12–20% slower than FlashGraph because the sk2005 graph has a high locality [14] such that storage access is minimized by hitting the page cache implemented in FlashGraph with LRU policy. Blaze only implements the random eviction of IO buffer pages, and we leave implementing more advanced eviction policies as future work.

On the other hand, Blaze consistently outperforms Graphene with 1.6–7.9$\times$ of speedups on our target workloads. In case of PR, we compare the execution time of 1 PR iteration as Graphene does not implement PR with selective scheduling.

5.4.3 IO Utilization

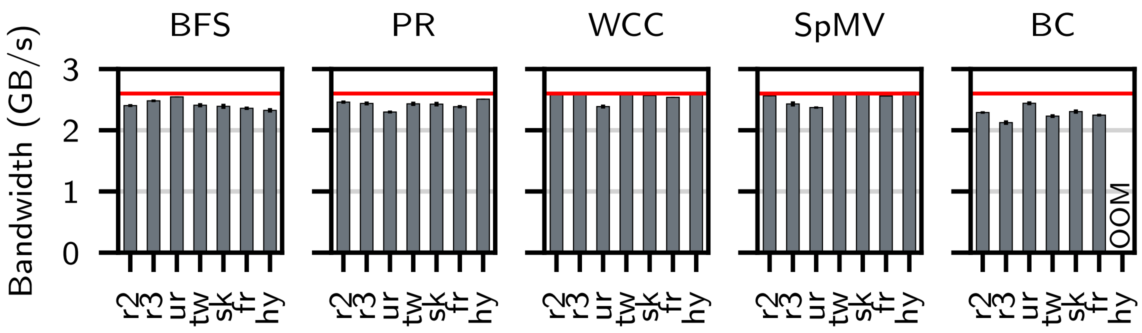
Blaze achieves high IO bandwidth close to the FND's device bandwidth. In Figure 5.8, we report the average IO bandwidth of our target workloads measured on Optane SSD. We calculate the average bandwidth as the total read IO bytes divided by the total query execution time. To see the impact of online binning on IO utilization, we compare Blaze with a synchronization-based variant of Blaze that uses atomic operations like `compare-and-swap` to synchronize parallel updates.



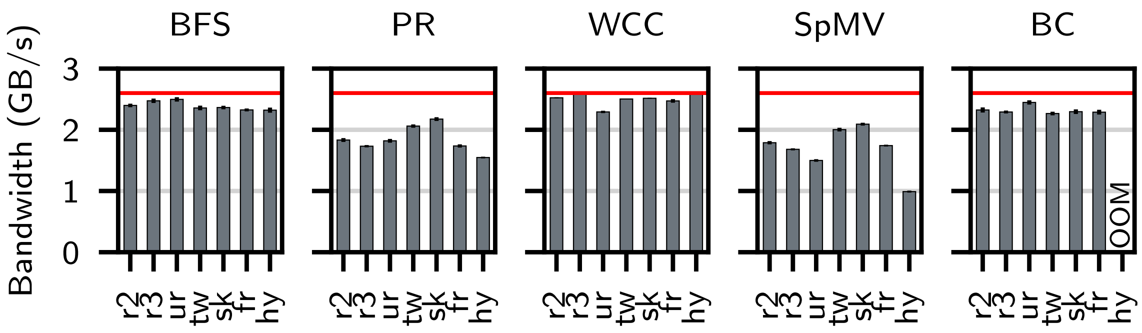
	BFS	PR	WCC	SpMV	BC
r2	1.24	7.74	3.92	1.85	2.18
r3	1.29	13.59	6.22	2.69	2.91
ur	1.23	1.64	7.24	2.42	1.42
tw	1.17	1.44	3.06	1.28	1.11
sk	0.81	0.88	3.52	0.85	0.81
fr	1.41	0.92	7.88	1.61	1.49

	BFS	PR	WCC	SpMV
r2	3.28	2.58	2.39	3.27
r3	3.57	1.64	2.35	1.63
ur	3.54	1.97	2.16	2.56
tw	4.05	2.09	5.25	1.62
sk	3.21	2.73	5.17	4.45
fr	4.41	2.22	7.95	2.19

Figure 5.7. Speedups over FlashGraph and Graphene.



(a) Blaze



(b) Synchronization-based variant

Figure 5.8. Average read bandwidth on Optane SSD

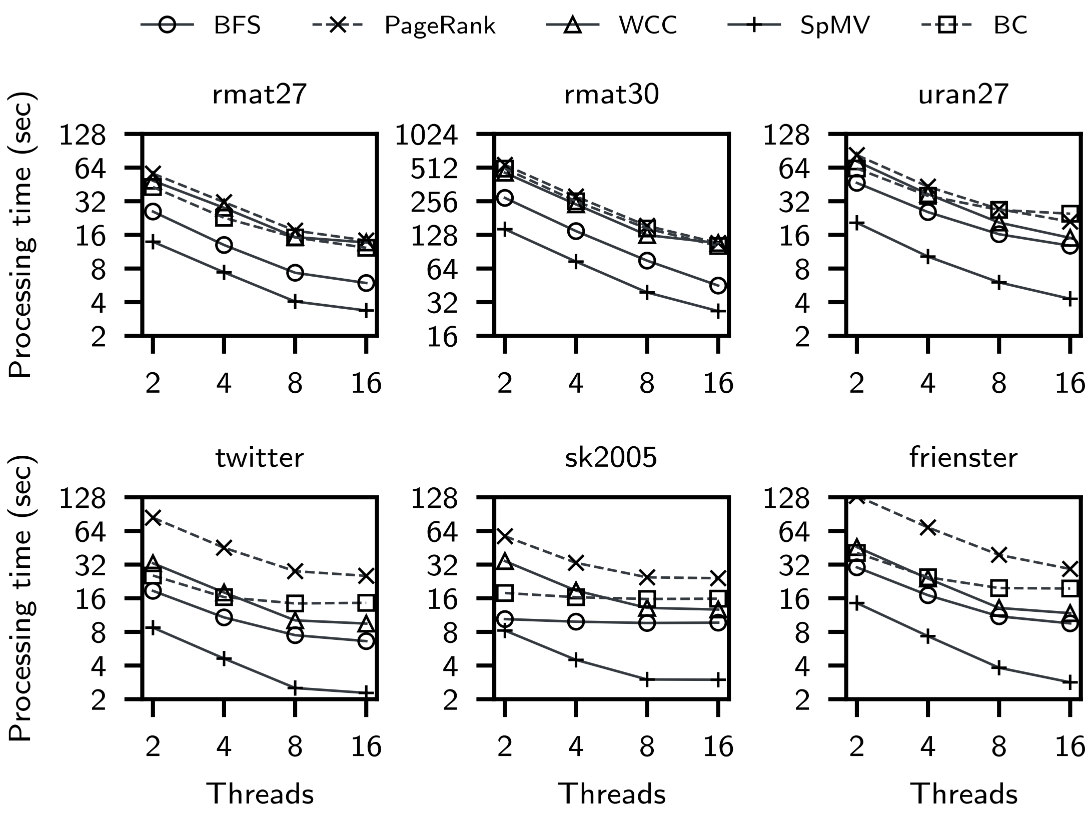


Figure 5.9. Thread scaling

Unlike FlashGraph and Graphene which underutilize an Optane SSD's bandwidth as reported in Figure 5.1, Blaze almost fully utilizes the bandwidth on all our workloads. On computation-heavy workloads like PageRank and SpMV, the high IO bandwidth is only achieved with online binning. Otherwise, the synchronization-based Blaze achieves only 38–85% of the Optane's device bandwidth on both queries depending on the workload.

5.4.4 Scalability

Blaze scales with increasing core count as long as the underlying storage is not saturated. Figure 5.9 (with both axes in log scale) shows how Blaze's performance scales when running our workloads on a single Optane SSD. Performance almost linearly

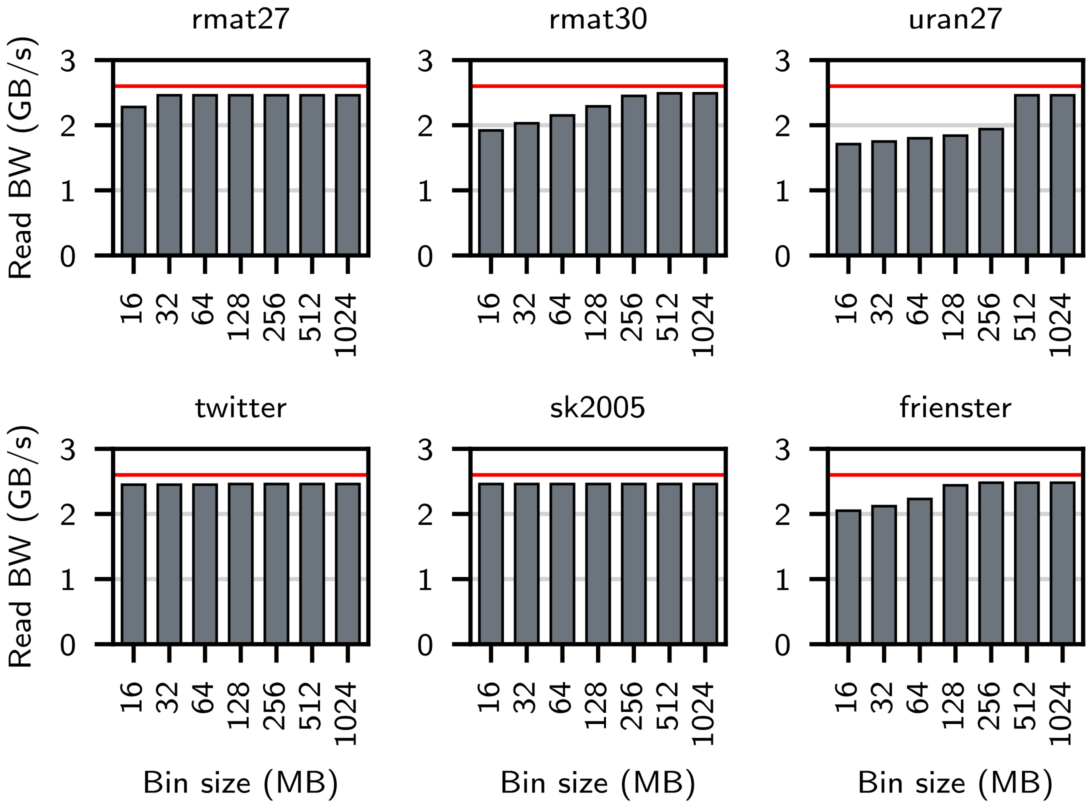


Figure 5.10. Impact of binning space.

scales with more cores on most of the workloads. On a certain set of workloads (e.g., BFS on sk2005), using one scatter and one gather thread (therefore two cores) is sufficient to saturate the IO bandwidth as the graph has high locality and thus causes less CPU overhead with processor cache hits. In these cases, Blaze does not scale with more threads as the IO bandwidth becomes the bottleneck.

5.4.5 Impact of Online Binning Configurations

Online binning in Blaze requires users to set up a few parameters to perform as expected.

Bin size The total bin size must be set large enough not to slow down the performance of Blaze. To understand the right bin size, we measured the average read bandwidth of

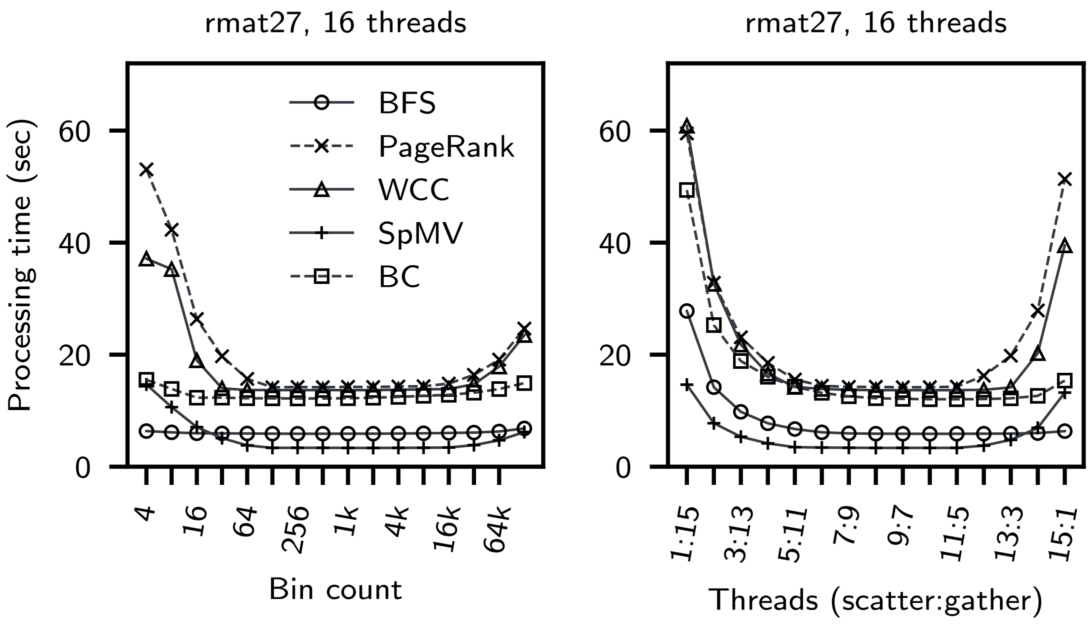


Figure 5.11. Impact of binning configurations.

SpMV query on all input graphs while varying the total bin size from 16 MB to 1 GB. Based on the result in Figure 5.10, we find that a good heuristic value for the bin size is roughly $\frac{1}{20}|E| \times 4$ *bytes* for each graph but using smaller values is also viable on some graphs.

Bin count We also study how different bin counts can impact the performance of online binning. For this, we measure the processing time of all queries on rmat27 graph with 256 MB of bin space while doubling the bin count from 4 to 131072. Figure 5.11 shows that the processing time is relatively stable for a large range of bin counts but increases significantly when the value is too large or too small.

Scatter, gather thread ratio An intuition in choosing the right ratio between the number of scatter and gather threads is to consider the relative computation load between scatter and gather in each algorithm. If the load is similar, a good choice is to use an equal number of threads for both tasks. The result in Figure 5.11 reflects this intuition – The execution time remains constantly low when a similar amount of threads

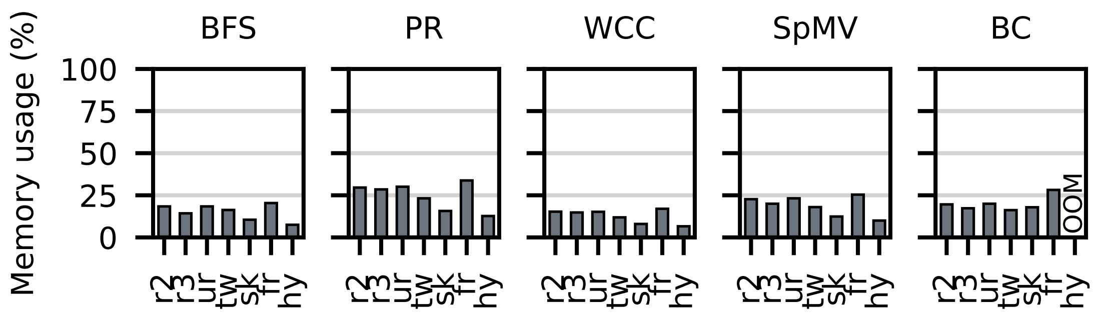


Figure 5.12. Memory footprint relative to the input graph size.

are used for scatter and gather but sharply increases as more threads are assigned for one task than the other.

5.4.6 Memory Usage

Under the semi-external model, Blaze aims to minimize the memory consumption while supporting high-performance out-of-core graph processing on FNDs. Figure 5.12 shows the memory consumption of Blaze on our target workloads. The figure depicts the ratio of memory footprint to input graph size in each workload.

Depending on the workload, Blaze's memory footprint is 10–34% of the input graph size. Concretely, memory footprint is affected by the input query and the underlying graph. For BFS that requires only a single integer array to keep the parent relationships (Algorithm 1), the ratio is 10–20% throughout different input graphs.

However, certain queries require more memory due to the nature of their algorithms. For instance, in case of our PageRank implementation, the ratio goes up to 16–33% as PageRank-delta requires three floating point arrays to implement (Algorithm 2). In addition, we were not able to run BC on hyperlink2014 graph because Brandes's algorithm [21] requires more than 96 GB of memory to run on 512 GB of undirected hyperlink2014 graph. We expect mitigating these problems require more memory-efficient algorithms.

Except for BC, Blaze can successfully run other queries on hyperlink2014 with the limited amount of memory compared to the graph size while existing in-core frameworks [96, 78, 119] will run into the out-of-memory issue with this dataset.

5.5 Related Work

The design of graph processing systems has been evolved mainly in three different ways depending on the use of secondary storage and the use of memories in multiple machines.

In-core graph processing processes graphs entirely in the main memory of a single machine. Galois [78] is a lightweight runtime that offers API for implementing efficient task scheduling of various graph algorithms. GAP [15] offers a benchmark suite of various in-core graph algorithms to standardize the in-core graph processing evaluations. GraphIt [119] introduces a new domain-specific language and runtime to help separate the process of algorithm writing and performance optimization. Finally, Ligra [96] is an in-core graph processing framework that offers simple APIs for writing graph algorithms and optimizes the execution of those algorithms by automatically switching between push and pull-based operations based on a user-provided threshold. Blaze extends the Ligra's APIs but adapts them to support efficient out-of-core execution.

Out-of-core graph processing uses storage devices to hold the large graphs that do not fit in a single machine's memory and process those graphs by judiciously moving pages between the storage and the memory. To minimize the overhead of expensive IO, the design of out-of-core systems has evolved in step with the performance characteristics of underlying storage devices. GraphChi [64] was the first out-of-core graph processing system designed for sequential disks. XStream [89] explored a new tradeoff that fully exploits the benefit of sequential access at the cost of more IO. More out-of-core systems have been further developed to leverage the sequentiality of disks [123, 52, 108].

On the other hand, systems like FlashGraph [121] and Graphene [67] explored the ways to utilize fast random IO that early-generation SSDs offer, achieving significant performance improvement over prior systems as less IO is required. Blaze makes a similar design choice but further optimizes software mechanisms to constantly saturate the high IO bandwidth of modern FNDs. A more recent work by Elyasi *et al.* [37] explored a new graph partitioning technique to leverage the fast random IO of FNDs in the fully-external processing style. However, this work makes a different tradeoff from Blaze – it places only a subset of vertex data in memory to achieve smaller memory footprint but potentially at the cost of performance. Unfortunately, we could not compare this work with Blaze as it is not public.

Distributed graph processing is another way to process large graphs by holding them in the memories of multiple machines and processing them via network communications. Systems like PowerGraph [43], Pegasus [56], and GraphLab [69] have been developed to ease the programming of distributed graph algorithms while offering high performance on highly-skewed graphs. Nonetheless, distributed graph processing systems often lack efficiency with low per-CPU performance [72] and do not necessarily outperform out-of-core systems despite using more resources from a cluster of machines [64, 123, 121, 67].

For higher scalability in terms of both CPU and storage, Blaze could be further scaled out on multiple machines where each machine is equipped with one or more FNDs. One potential way to scale out Blaze is to partition the input graph based on the *destination vertex* and place each partition in each machine. This allows a single machine to process only a subset of edges and vertex-related values, and, more importantly, to propagate values between scatter and gather threads *locally*, avoiding the costly network communications during EDGEMAP execution. We leave scaling out Blaze in this manner as future work.

5.6 Summary

We present Blaze, a new out-of-core graph processing system optimized for FNDs. Blaze offers high-performance graph analytics by constantly saturating FNDs with a novel scatter-gather technique called online binning while previous techniques like synchronization or message passing fail to achieve this goal. Blaze offers succinct APIs for writing efficient, out-of-core graph algorithms without the burden to deal with complex IO executions.

Acknowledgements

This chapter contains material from “Blaze: Fast Graph Processing on Fast SSDs”, by Juno Kim and Steven Swanson, which appeared in the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2022). The dissertation author is the primary investigator and the first author of this paper.

Chapter 6

TOSS: Tiering of Serverless Snapshots for Low Cost Serverless Computing

Serverless computing is a new cloud computing paradigm that simplifies the programming of cloud computing – A user defines only the desired function and delegates its execution to the cloud by simply submitting the defined function to the provider. Due to its simplicity and promises, serverless computing is expected to prosper with various offerings from major cloud providers [90].

One of the key challenges about instantly executing a serverless function is to quickly prepare a function instance (i.e., container, VM) in memory. To solve this problem, a slew of techniques have been proposed [104, 42], all of which aim to prepare necessary memory contents of a function instance on DRAM. However, using only a single tier of expensive memory incurs high cost and does not expand well beyond a certain capacity, both of which lead to low total cost of ownership (TCO) for serverless providers.

Unfortunately, existing memory tiering approaches are not suitable for serverless computing. First, serverless functions' execution expose unique patterns. Second, they are short-running, with execution times that span mostly to several seconds [94], usually with a small memory footprint. Finaly, they are infrequently but repeatedly invoked until removed. These characteristics impose new challenges for existing tiering techniques, as the latter aim to simply evict the colder memory to the slower tiers; in a serverless environment that would lead to significant cold-start delays for infrequently-called serverless functions.

In this work, we propose TOSS, the first two-tier serverless system that aims to reduce the memory costs of a serverless platform that leverages only DRAM. The key idea is to offload each function's certain memory pages to the slower tier, while minimizing the target function's slowdown and making it under a certain boundary. To achieve this, TOSS introduces *performance-driven serverless snapshot tiering* where each function's slowdown caused by memory tiering is bounded by a user-defined performance goal. We build the TOSS prototype on top of Firecracker's MicroVM snapshotting mechanism.

6.1 Background and Motivation

6.1.1 Serverless Computing

Serverless computing is a new cloud computing paradigm that allows developers to focus on building and deploying applications on infrastructure that is managed by cloud providers. In serverless computing, developers only pay for the resources they use, while it is easier to scale applications, as the infrastructure provider adjusts the load automatically to meet demand. Serverless computing helps organizations to be more efficient, agile, and cost-effective, and is an essential part of modern cloud-native architectures.

From the user's perspective, serverless computing offers several benefits. First, users only pay for the resources their code is consuming, which can lead to significant cost savings compared to traditional cloud computing models. Second, users can quickly and easily deploy their code without worrying about managing servers. This allows them to be more flexible and responsive to their business needs. Third, the infrastructure automatically adjusts to meet demand, making it easier to scale applications and services as needed.

On the other hand, serverless computing greatly benefits the cloud provider as well. First, providers can optimize the use of resources by automatically allocating them to different customers based on demand, which can lead to improved efficiency and utilization. Second, providers can more easily manage and maintain the infrastructure, which can lead to improved reliability and uptime for customers. Third, providers can offer a wide range of serverless computing options to customers, which can help to attract and retain a larger customer base.

6.1.2 Firecracker as Function Instance

Firecracker [9] is a popular virtualization technology that supports serverless computing and other cloud-native applications. It was developed by AWS and allows users to run multiple lightweight virtual machines on a single host. With its microVM having their own isolated operating system and resources, Firecracker allows fast and efficient cloud deployment, incurring minimal overhead and small memory and disk footprint. It is well-suited for running short-lived, stateless functions in a scalable and cost-effective fashion. Many organizations are adopting Firecracker as a key part of their serverless computing infrastructure to improve performance, security, and resource utilization.

In addition, Firecracker offers the snapshotting feature [41] to allow users to create, save, and restore the state of microVMs instantly. Snapshots are created using copy-on-write on the root filesystem, which means they only store the differences between the current state and the previous snapshot, which helps to keep the snapshot size small. To restore a snapshot, Firecracker copies the snapshot data back to the root filesystem, overwriting any changes made since the snapshot was taken. The snapshotting mechanism is useful for managing the state of microVMs and quickly booting new microVMs or reverting to previous states.

6.1.3 Memory Tiering

In a system with two memory tiers, fast memory (such as DRAM) and slow memory (such as persistent memory or CXL memory), memory tiering can be used to reduce the memory cost of computing. Fast memory provides a higher access speed but it is more expensive and limited in capacity while slow memory has an opposite tradeoff. Memory tiering allows the system to keep the most frequently accessed data in fast memory while less frequently accessed data is offloaded to slow memory. This helps to keep the performance fast while expanding the memory capacity with low cost.

6.1.4 Motivation

Despite recent progress for serverless computing on providing fast startup of VMs, current serverless offerings are limited by the sole use of fast memory, which incurs high memory cost and limits the capacity scaling of memory. While memory tiering is an active research area to solve this problem, this idea has not been explored in serverless computing environment. Therefore, the motivation of this work is to apply the memory tiering idea to serverless computing to offer a more cost-efficient serverless execution environment enabled by an active use of slow memory tier (such as CXL memory).

6.2 TOSS

TOSS is the first serverless system that supports memory tiering for serverless functions. It offers both fast VM startups and fast function execution, while maintaining a low cost approach, compared to a pure-DRAM approach. TOSS creates a tiered VM snapshot, placing pages on different memory tiers, depending on their access count. This helps maximizing the use of slow memory, while meeting each function's

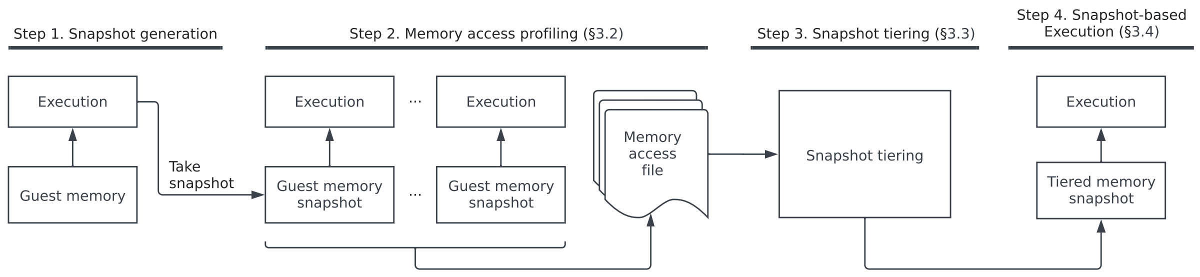


Figure 6.1. VM snapshot tiering steps in TOSS

performance goal. A stricter performance goal limits TOSS from offloading more pages to the slower memory tier. We demonstrate how different performance goals impact memory tiering in TOSS in Section 6.3.

6.2.1 Overview

TOSS supports memory tiering of serverless functions in four steps as depicted in Figure 6.1. First, TOSS executes a function in DRAM-only guest VM and takes a VM snapshot after the execution finishes. This initial snapshot will later be used to create the tiered memory snapshot that will be split in different memory tiers. Second, TOSS enters the memory access profiling step, where it gathers memory access information for subsequent invocations (Section 6.2.2). The information is collected for each invocation to capture divergent memory access behavior that different invocations can exhibit. Third, TOSS offloads snapshot pages to the slow memory tier based on the memory access information and its performance estimation method (Section 6.2.3). We use the term *tiered snapshot* to describe the multi-tier snapshot that TOSS leverages. Finally, TOSS serves subsequent function invocations with the tiered snapshot (Section 6.2.4). If there is a repeated violation of the user-provided performance goal, TOSS starts over the tiering process for the function.

6.2.2 Memory Access Profiling

TOSS relies on two memory access metrics for snapshot tiering, namely *memory stall ratio* and *memory access count*. Memory stall ratio is the memory stall time relative to the total execution time of a function, gives how much memory-intensive a function is, suggesting how much slowdown a function is likely to suffer when offloaded to slow memory. On the other hand, memory access temperature is a list of values where each element represents an access count of each memory page in a given function's snapshot. Memory access temperature helps TOSS identify cold pages – pages with low access count – for offloading to slow memory.

To collect these counters, TOSS uses both hardware and software-based techniques. To measure memory stall ratio of each function, TOSS collects the hardware counter that gives the number of cycles spent stalling on outgoing memory load requests. With the measured memory stall cycles, TOSS calculates memory stall ratio by dividing the value with the cycles spent during the execution of the function. In serverless environment, these cycle counters can easily be measured for each process using Linux's `perf-kvm` tool, so TOSS uses it for each Firecracker guest VM process.

TOSS uses Data Access MONitoring (DAMON) [93], an accurate, light-weight and scalable data access monitoring framework, to measure the memory access count of each function. DAMON leverages the Accessed bit that can be found in the Linux page table for each associated page. It clears periodically the Accessed bit, which is set when there is a page table walk due to a TLB miss. While this sampling approach gives an estimation, rather than the actual access count, it gives a good approximation on the relative access count among different pages that belong to the same process. Therefore, TOSS uses DAMON to profile the memory accesses of each serverless function.

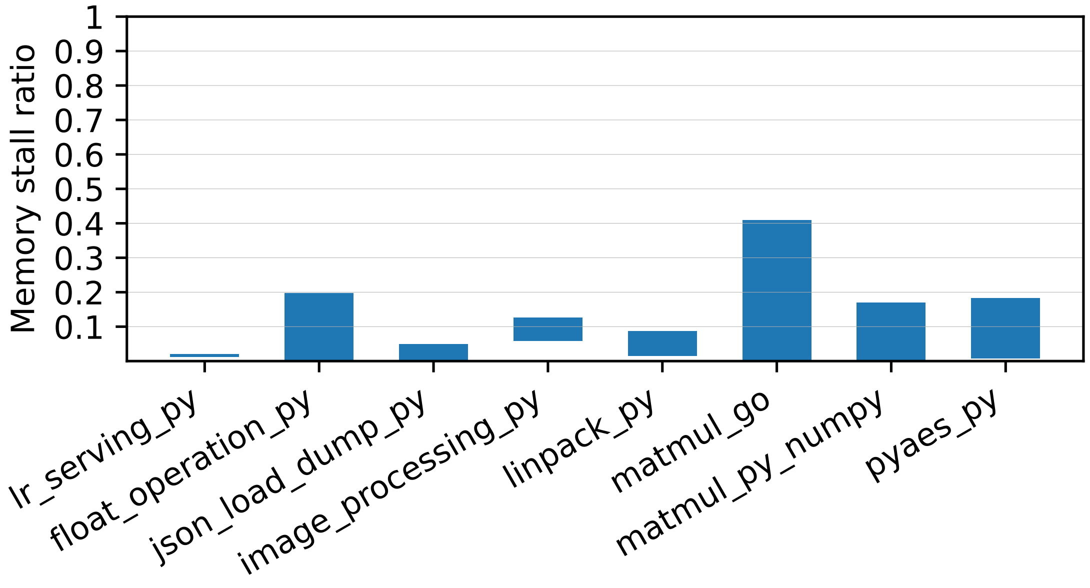


Figure 6.2. Memory stall ratio of target functions. The bottom and top of each bar correspond to the minimum and maximum of memory stall ratio, respectively, of each function based on varying inputs

6.2.3 Performance-driven Snapshot Tiering

Based on the memory access statistics collected over invocations of a function, TOSS distributes the function's snapshot pages across fast and slow memories. The goal of the tiered snapshot is to restrict the corresponding function's execution times under a certain slowdown to ensure that the use of slow memory does not incur too much performance penalty. To achieve this, TOSS considers three factors – *execution statistics*, *target slowdown*, and *performance traits of different memories* when selecting the target snapshot pages to be placed in slow memory. Based on these information, the snapshot tiering in TOSS is conducted in two steps, *slow memory access estimation* and *cold page selection*, which we describe in detail as follows.

Slow memory access estimation The first step is estimating the *slow memory access ratio* – the number of accesses to slow memory over the number of accesses to all memories – given a user's slowdown goal. TOSS estimates the slow memory access ratio by using the following formula:

$$R_{slow} = \frac{S_{func}}{S_{dev} \times M_{func}} \tag{6.1}$$

S_{func} is the *slowdown threshold* of a given function. A higher slowdown allows TOSS to move memory pages more aggresively to the slower tier.

S_{dev} is the *device-level slowdown*, which depicts the slowdown of the slow memory device related to the fast memory device. When S_{dev} is higher, the performance penalty by snapshot tiering becomes higher so TOSS must restrict the number of pages residing in the slow memory. This value depends on the type of fast and slow memory devices used for memory tiering. For instance, when the fast memory is DRAM and the slow memory is Optane™ DC Persistent Memory, the value of this term becomes 3, based on relative latencies between the two devices [115].

M_{func} is the *memory stall ratio* of a given function calculated as the fraction of the memory stall cycles over the total execution cycles. When a function is more memory-intensive (higher M_{func}), it is more susceptible to performance degradation when offloaded to the slow memory tier, implying that TOSS must move less memory pages to slow memory. We measure this in Section 6.2.2.

Once the slow memory access ratio R_{slow} is calculated, it effectively becomes the tiering goal that TOSS must achieve to satisfy the slowdown goal S_{func} given by the user.

Cold page selection TOSS selects cold pages in a way that it meets the slow memory access ratio R_{slow}. To achieve this, TOSS's cold page selection algorithm takes R_{slow} and memory access temperature *acc_list* (collected in Section 6.2.2) as input, and produces a set of cold pages P_{cold}. TOSS repeats this to calculate $P_{cold,i}$ from i^{th} invocation result. Finally, TOSS intersects $P_{cold,1}$, $P_{cold,2}$, ..., $P_{cold,n}$ to get the final cold page set. The set intersection ensures that the final result always becomes a subset of $P_{cold,i}$, ensuring that it does not violate the performance target with any input to the function.

Listing 6.1. Cold page selection algorithm

```
def cold_page_selection(R_slow, acc_list):
    # sort by access count in ascending order
    acc_list.sort(key = lambda x: x[1])

    # select as many pages as possible that meet R_slow
    total_acc_cnt = sum([cnt for _, cnt in acc_list])
    acc_cnt = 0

    P_cold= []
    for page, cnt in acc_list:
        acc_cnt += cnt
        r = float(acc_cnt) / total_acc_cnt
        if r > R_slow:
            break
        P_cold.append(page)

    return P_cold
```

The algorithm is described in Listing 6.1. TOSS first sorts the access count list *acc_list*, a list of ¡page number, access count¿ pairs, in ascending order based on the access count (line 3). Then it iterates over the sorted list and adds the page to the output set P_{cold}, as long as the accumulated access ratio r is slower than the given R_{slow} (line 6–15). By choosing the pages with the smallest access counts, TOSS maximizes the size of P_{cold} while meeting the given performance goal S_{func}.

6.2.4 Function Execution on Tiered Snapshot

Given the cold page set identified by the profiling results and the cold page selection algorithm, TOSS classifies hot and cold regions of each function's snapshot. One region indicates a set of contiguous pages. For each hot region, TOSS creates a file in fast memory. For each cold region, it creates a file in slow memory. In our TOSS prototype, we use Ext4 file system in DAX mode [111] and mount it in two different paths, one is backed by DRAM and the other is backed by PMem.

Once hot and cold files are generated, our modified Firecracker maps them into a single guest address space which consists of multiple guest memory regions. This is easily supported by `vm-memory` substrate in Rust used by the standard Firecracker implementation. We leverage it to implement tiered memory snapshot in TOSS. Once the guest memory space is created on top of the tiered memory regions, microVMs loaded with that memory space run as usual while the underlying memory is now backed by both fast and slow memory.

Table 6.1. TOSS target functions.

Name	Description	Language	Guest VM Memory Size
lr_serving	Logistic regression inferencing	Python	1024 MB
json_load_dump	Read-Modify-Write JSON files	Python	128 MB
image_processing	Flips the input image	Python	256 MB
linpack	Solves the linear equation Ax = b	Python	256 MB
matmul	Generates the matrix product	Python, Go	256 MB
pyaes	AES Text encryption	Python	128 MB
rma	Synthetic random memory access	Go	128 MB

6.3 Evaluation

We evaluate the performance and effectiveness of TOSS by answering the following questions.

- How much memory TOSS can offload to slow memory while meeting the performance goal?
- How effectively does TOSS support memory tiering under various functions?

To answer these, we use a representative set of function workloads used by prior works [59] as specified in Table 6.1. We additionally write the matrix multiplication function, `matmul`, in Go to see the impact of using a different programming language to serve the same function. Moreover, we evaluate a synthetic function, `rma`, that incurs heavy, random memory accesses to see how TOSS performs even with functions with little memory access locality. We use different guest VM memory size to best serve each function's memory needs.

Under this environment, we evaluate TOSS with two different performance target, 5% and 10%, as depicted in Figure 6.3 and Figure 6.4, respectively. The result indicates TOSS can offload a significant amount of memory to slow memory while achieving its performance goal. In its best case, TOSS moves more than 90% of memory

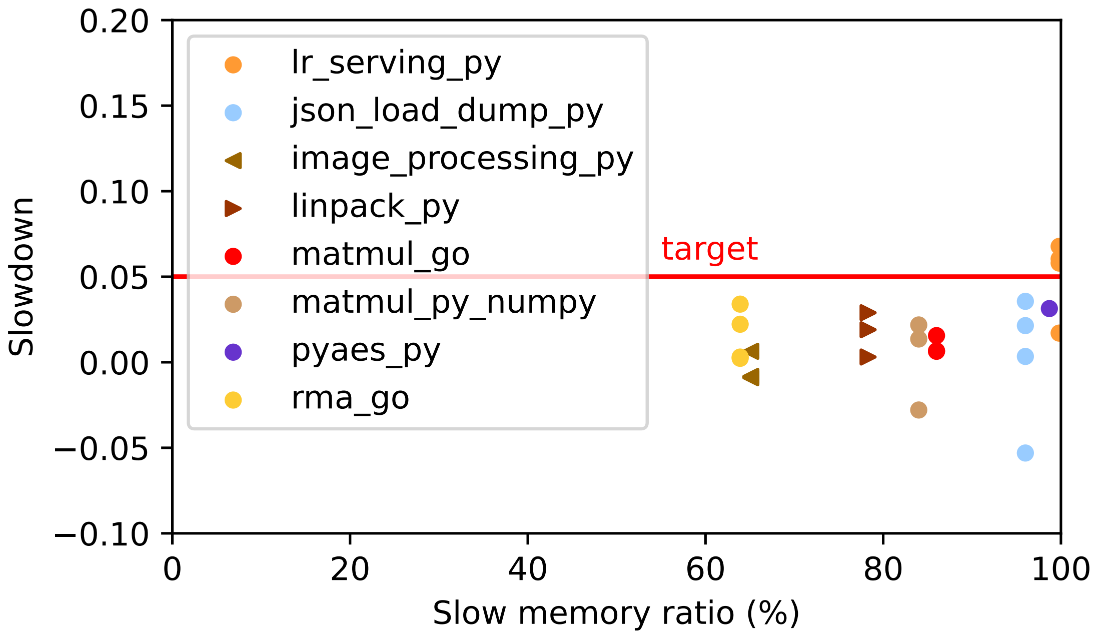


Figure 6.3. Functions on tiered snapshots under 5% of slowdown goal.

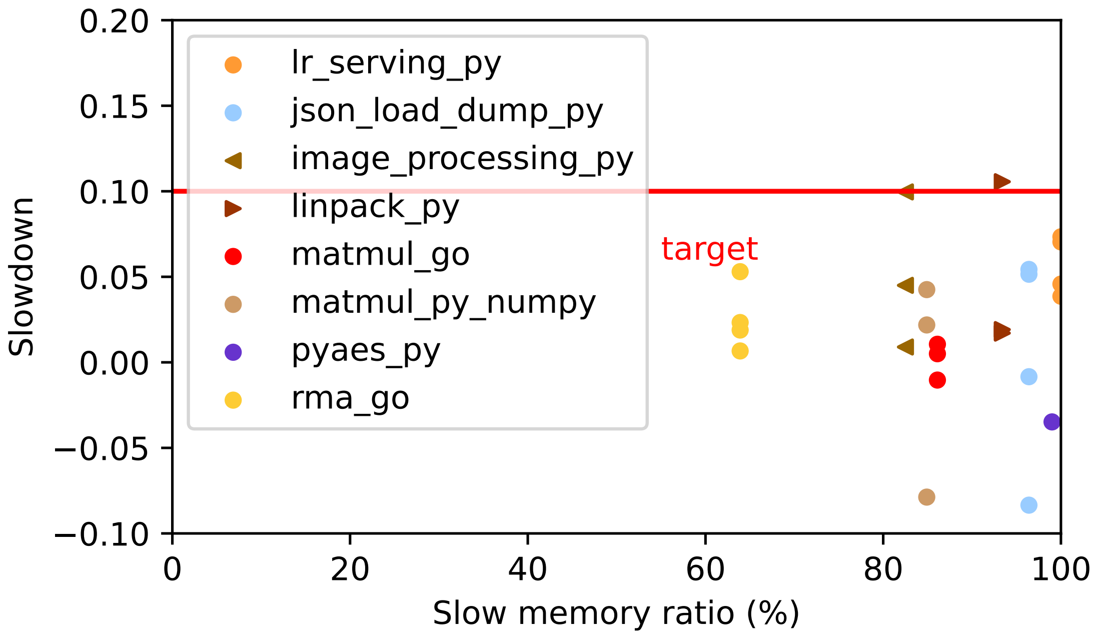


Figure 6.4. Functions on tiered snapshots under 10% of slowdown goal.

to the slow tier (`lr_serving_py`, `pyaes_py`, `json_load_dump_py`) under both 5% and 10% targets. In its worst case, TOSS still moves about 65% of memory to the slow tier (`rma_go`) while meeting the performance goals. TOSS achieves this for various inputs to the same function (different dots in the same color indicate different inputs to the same function).

6.4 Summary

In this work, we present TOSS, the first platform that helps reduce the memory cost of serverless computing by leveraging memory tiering. To support this, TOSS newly introduces *tiered snapshot mechanism* that takes the best of both microVM snapshotting and memory tiering. Our evaluation shows that TOSS is able to offload a significant amount of memory into the slow memory tier while keeping the potential slowdown of functions under control.

Acknowledgements

This chapter contains material from "TOSS: Tiering of Serverless Snapshots for Low Cost Serverless Computing", by Juno Kim, Theodoros Michailidis, Linsong Guo, Jishen Zhao, and Steven Swanson, which is in preparation for submission to the European Conference on Computer Systems (EuroSys 2024). The dissertation author is the primary investigator and the co-first author of this paper.

Chapter 7

Conclusion

Emerging memory technologies, such as persistent memory and ultra-low-latency SSDs, change the tradeoffs system designers must consider. As storage, they offer considerably faster persistence than traditional SSDs and hard disks. As memory, they expand the capacity of main memory in a single machine at a lower price than conventional, expensive DRAM. Despite these advantages, efficiently leveraging them remains challenging in a variety of software stacks including file systems, databases, graph processing systems, and serverless computing.

Throughout this dissertation, we have presented software techniques that best leverage emerging memory technologies in various software stacks. These techniques enable data-intensive applications and system software to fully utilize the performance, capacity, direct-accessibility of emerging memories with minimized burden to programmers. The techniques we introduced include general solutions implemented in system software layer such as file system as well as domain-specific solutions aimed for user applications such as databases or graph processing systems. In addition, the dissertation described how emerging memories can be used to lower the cost of datacenter applications by supporting memory tiering in the serverless computing stack.

In Chapter 3, we have examined the performance of NVMM storage software stacks to identify the bottlenecks and understand how both applications and the operating system should adapt to exploit NVMM performance.

We examined several applications and identified several simple techniques that provide significant gains. The most widely applicable of these use FLEX to move writes to user space, but implementing `msync` in userspace and assiduously avoiding metadata operations also help, especially on adapted NVMM file systems. Notably, our results show that FLEX can deliver nearly the same level of performance as building crash-consistent data structures in NVMM but with much less effort.

On the file system side, we evaluated solutions to the problems of inefficient logging in adapted NVMM file systems, multicore scaling limitations in file systems and the Linux's VFS layer, and the novel challenge of dealing with NUMA effects in the context of NVMM storage.

Overall, we find that although there are many opportunities for further improvement, the efforts of systems designers over the last several years to prepare systems for NVMM have been largely successful. As a result, there are a range of attractive paths for legacy applications to follow as they migrate to NVMM.

In Chapter 4, we have described and implemented SUBZERO, a new IO mechanism that avoids most or all data movement for reads and writes to PMEM-backed files. In addition to minimizing movement, our implementation of SUBZERO provides both fast read access and strongly consistent updates. Our evaluation shows that SUBZERO outperforms copy-based `read()` and `write()` by a wide margin. In summary, SUBZERO IO is a straight-forward way for programmers to improve their applications' performance on PMEM file systems.

In Chapter 5, we present Blaze, a new out-of-core graph processing system optimized for FNDs. Blaze offers high-performance graph analytics by constantly saturating FNDs with a novel scatter-gather technique called online binning while previous techniques like synchronization or message passing fail to achieve this goal. Blaze offers succinct APIs for writing efficient, out-of-core graph algorithms without the burden to deal with complex IO executions.

In Chapter 6, we have presented TOSS, the first platform that helps reduce the memory cost of serverless computing by leveraging memory tiering. To support this, TOSS newly introduces *tiered snapshot mechanism* that takes the best of both microVM snapshotting and memory tiering. Our evaluation shows that TOSS is able to offload a significant amount of memory into the slow memory tier while keeping the potential slowdown of functions under control.

Acknowledgements

This chapter contains material from “Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks”, by Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson, which appeared in the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019). The dissertation author is the primary investigator and the co-first author of this paper.

This chapter contains material from “SubZero: Zero-Copy IO for Persistent Main Memory File Systems”, by Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson, which appeared in the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2020). The dissertation author is the primary investigator and the first author of this paper.

This chapter contains material from “Blaze: Fast Graph Processing on Fast SSDs”, by Juno Kim and Steven Swanson, which appeared in the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2022). The dissertation author is the primary investigator and the first author of this paper.

This chapter contains material from “TOSS: Tiering of Serverless Snapshots for Low Cost Serverless Computing”, by Juno Kim, Theodoros Michailidis, Linsong

Guo, Jishen Zhao, and Steven Swanson, which is in preparation for submission to the European Conference on Computer Systems (EuroSys 2024). The dissertation author is the primary investigator and the co-first author of this paper.

Bibliography

[1] Apache bench (ab) - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/en/programs/ab.html.

[2] Extent swap in Ext4. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[3] Intel optane ssd 9 series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[4] A programmer's guide to the mach system calls. http://shakthimaan.com/downloads/hurd/A.Programmers.Guide.to.the.Mach.System.Calls.pdf.

[5] Samsung v-nand ssd. https://www.samsung.com/us/business/computing/memory-storage/solid-state-drives/explore/.

[6] Samsung z-ssd. https://semiconductor.samsung.com/ssd/z-ssd/.

[7] CVE-2010-2066, 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2066.

[8] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[9] 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.

[10] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.*, 10(12):1754–1765, August 2017.

[11] Apple File System, 2017. https://en.wikipedia.org/wiki/Apple_File_System.

[12] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 11(5):553–565, January 2018.

[13] S. Beamer, K. Asanović, and D. Patterson. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831, 2017.

[14] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, IISWC '15, page 56–65, USA, 2015. IEEE Computer Society.

[15] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[16] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 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, pages 677–694, New York, NY, USA, 2016. ACM.

[17] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. Bankshot: Caching Slow Storage in Fast Non-volatile Memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 1:1–1:9, New York, NY, USA, 2013. ACM.

[18] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, New York, NY, USA, 2017. ACM.

[19] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[20] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[21] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[22] Matthew J. Breitwisch. Phase change memory. *Interconnect Technology Conference, 2008. IITC 2008. International*, pages 219–221, June 2008.

[23] Adrian M. Caulfield, Todor I. Mollov, Louis Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceeding of the 17th international conference on Architectural support for programming languages and operating systems*. ACM, March 2012.

[24] 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.

[25] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Optimizing File Systems with Fine-grained Metadata Journaling on Byte-addressable NVM. *ACM Trans. Storage*, 13(2):13:1–13:25, May 2017.

[26] Jianxi Chen, Qingsong Wei, Cheng Chen, and Lingkun Wu. FSMAC: A file system metadata accelerator with non-volatile memory. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–11. IEEE, 2013.

[27] Dave Chinner. xfs: updates for 4.2-rc1, 2015. http://oss.sgi.com/archives/xfs/2015-06/msg00478.html.

[28] Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, Junho Shin, Yoohwan Rho, Changsoo Lee, Min Gu Kang, Jaeyun Lee, Yongjin Kwon, Soehee Kim, Jaehwan Kim, Yong-Jun Lee, Qi Wang, Sooho Cha, Sujin Ahn, H. Horii, Jaewook Lee, Kisung Kim, Hansung Joo, Kwangjin Lee, Yeong-Taek Lee, Jeihwan Yoo, and G. Jeong. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 46–48, Feb 2012.

[29] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4):10:1–10:47, January 2015.

[30] 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.

[31] 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.

[32] 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. ACM, 2010.

[33] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for nvrams. *Proc. VLDB Endow.*, 13(9):1598–1613, May 2020.

[34] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[35] Mingkai Dong, Qianqian Yu, Xiaozhou Zhou, Yang Hong, Haibo Chen, and Binyu Zang. Rethinking benchmarking for NVM-based file systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 20:1–20:7, New York, NY, USA, 2016. ACM.

[36] 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 15:1–15:15, New York, NY, USA, 2014. ACM.

[37] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 309–316, USA, 2019. USENIX Association.

[38] Facebook. RocksDB, 2017. http://rocksdb.org.

[39] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush. A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 338–339, Feb 2014.

[40] FAL Labs. Kyoto Cabinet: a straightforward implementation of DBM, 2010. http://fallabs.com/kyotocabinet/.

[41] Firecracker authors. Firecracker snapshotting. https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md.

[42] 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 2021, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association.

[44] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, page 77–85, New York, NY, USA, 2013. Association for Computing Machinery.

[45] Robin Harris. Windows leaps into the NVM revolution, 2016. http://www.zdnet.com/article/windows-leaps-into-the-nvm-revolution/.

[46] Hewlett Packard Enterprise. HPE Scalable Persistent Memory, 2018. https://www.hpe.com/us/en/servers/persistent-memory.html.

[47] Intel. NVDIMM Namespace Specification, 2015. http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf.

[48] Intel. Intel Architecture Instruction Set Extensions Programming Reference, 2017. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[49] Silicon Graphics International. XFS: A High-performance Journaling Filesystem. http://oss.sgi.com/projects/xfs.

[50] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. AndroStep: Android Storage Performance Analysis Tool. In *Software Engineering (Workshops)*, volume 13, pages 327–340, 2013.

[51] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 309–320, San Jose, CA, 2013. USENIX.

[52] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 411–424. IEEE Press, 2018.

[53] 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*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Dong Hyun Kang, Gihwan Oh, Dongki Kim, In Hwan Doh, Changwoo Min, Sang-Won Lee, and Young Ik Eom. When address remapping techniques meet consistency guarantee mechanisms. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association.

[55] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, Santa Clara, CA, 2015. USENIX Association.

[56] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238, 2009.

[57] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable numa-aware blocking synchronization primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 603–615, Santa Clara, CA, 2017. USENIX Association.

[58] Takayuki Kawahara. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *Design & Test of Computers, IEEE*, 28(1):52–63, Jan 2011.

[59] Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 477, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 385–398, New York, NY, USA, 2016. ACM.

[61] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast nvm storage with udepot. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 1–15, USA, 2019. USENIX Association.

[62] Youngjin Kwon. Personal communication, 2018.

[63] 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, pages 460–477, New York, NY, USA, 2017. ACM.

[64] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association.

[65] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[66] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 235–247, Santa Clara, CA, 2015. USENIX Association.

[67] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, February 2017. USENIX Association.

[68] Ran Liu, Heng Zhang, and Haibo Chen. Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, 2014. USENIX Association.

[69] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[70] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[71] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[72] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 14, USA, 2015. USENIX Association.

[73] 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, New York, NY, USA, 2017. ACM.

[74] Micron. 3D XPoint Technology, 2017. http://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[75] Micron. Hybrid Memory: Bridging the Gap Between DRAM Speed and NAND Nonvolatility, 2017. http://www.micron.com/products/dram-modules/nvdimm.

[76] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, 2016. USENIX Association.

[77] MongoDB, Inc. MongoDB, 2017. https://www.mongodb.com.

[78] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[79] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita. A 3.3ns-access-time 71.2uW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture. In *Solid-State Circuits Conference (ISSCC), 2015 IEEE International*, pages 1–3, Feb 2015.

[80] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. SQLite Optimization with Phase Change Memory for Mobile Applications. *Proc. VLDB Endow.*, 8(12):1454–1465, August 2015.

[81] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. Share interface in flash storage for relational and nosql databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 343–354, New York, NY, USA, 2016. ACM.

[82] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow.*, 10(11):1166–1177, August 2017.

[83] 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, Santa Clara, CA, 2017. USENIX Association.

[84] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

[85] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[86] S. Raoux, G.W. Burr, M.J. Breitwisch, C.T. Rettner, Y.C. Chen, R.M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L Lung, and C.H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[87] redislabs. Redis, 2017. https://redis.io.

[88] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[89] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[90] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, apr 2021.

[91] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. In *ADMS@ VLDB*, pages 61–72, 2015.

[92] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-atomic slotted paging for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 91–104, New York, NY, USA, 2017. ACM.

[93] SeongJae Park. DAMON: Data Access Monitor. https://sjp38.github.io/post/damon/.

[94] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[95] Julian Shun. *An Evaluation of Parallel Eccentricity Estimation Algorithms on Undirected Real-World Graphs*, page 1095–1104. Association for Computing Machinery, New York, NY, USA, 2015.

[96] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[97] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 227–240, Oakland, CA, 2018. USENIX Association.

[98] SQLite. SQLite, 2017. https://www.sqlite.org.

[99] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, page 540–546, New York, NY, USA, 2018. Association for Computing Machinery.

[100] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[101] Symas. Lightning Memory-Mapped Database (LMDB), 2017. https://symas.com/lmdb/.

[102] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[103] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to Get More Value from Your File System Directory Cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 441–456, New York, NY, USA, 2015. ACM.

[104] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. *Benchmarking, Analysis, and Optimization of Serverless Function Snapshots*, page 559–572. Association for Computing Machinery, New York, NY, USA, 2021.

[105] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, San Jose, CA, USA, February 2011. USENIX Association.

[106] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[107] 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.

[108] Keval Vora. LUMOS: Dependency-driven disk-based graph processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, Renton, WA, July 2019. USENIX Association.

[109] Dejan Vučinić, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojević, Luiz Franca-Neto, Damien Le Moal, Trevor Bunker, Jian Xu, Steven Swanson,

and Zvonimir Bandić. DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 309–315, Santa Clara, CA, 2014. USENIX.

[110] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Anvil: Advanced virtualization for modern non-volatile memory devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 111–118, Santa Clara, CA, 2015. USENIX Association.

[111] Matthew Wilcox. Add support for NV-DIMMs to ext4, 2014. https://lwn.net/Articles/613384/.

[112] 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, Santa Clara, CA, February 2016. USENIX Association.

[113] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[114] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memories and rdma-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 221–234, USA, 2019. USENIX Association.

[115] 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.

[116] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 3–3, Berkeley, CA, USA, 2012. USENIX.

[117] 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 *13th USENIX Conference on File and Storage Technologies*, FAST '15, pages 167–181, Santa Clara, CA, February 2015. USENIX Association.

[118] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.

In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[119] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[120] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[121] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.

[122] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, 2002.

[123] 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 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.

[124] Ross Zwisler. Add support for new persistent memory instructions. https://lwn.net/Articles/619851/.