Evolving System Stack for Persistent Memory: Device Characterization, Caching, and Sharing Perspectives

By

Kan Wu

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2022

Date of final oral examination: August 17, 2022

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Michael M. Swift, Professor, Computer Sciences
Shivaram Venkataraman, Professor, Computer Sciences
Kassem M. Fawaz, Professor, Electrical and Computer Engineering

To my father.

Acknowledgments

This thesis would not have been possible without the help and guidance of many people, for whom I am grateful.

First and foremost, I would like to express my heartfelt gratitude to Andrea and Remzi, my advisors. I liked their research styles and how they gave me advice. I like Remzi and Andrea's system research principle of "Measure, then Build." This is exactly how I completed my Ph.D.. It was the deep measurements that my advisors encouraged me to take during my early Ph.D. years so that I could gain a lot of insights into various systems and generate various ideas based on the knowledge. Andrea is very thoughtful about all aspects of my projects, including measurement, design, and writing. I appreciate Andrea's advice to always bring a measurement figure to our meetings to discuss. Working with Andrea aided me greatly as a researcher. I admire Remzi's ability to generalize concepts and ideas. As a researcher, he has always served as a role model for me. Also, from the perspective of a Ph.D., his ability to generalize things means that there are always things worth exploring, even if we failed on some ideas, which has made my Ph.D. life a lot easier. Andrea and Remzi put in a lot of effort on my research. They care about students and have always encouraged me throughout my Ph.D. journey, as well as given me freedom in research topics and time management. I honestly could not have asked for better advisors. Thank you so much, Andrea and Remzi!

I'd also like to thank my committee members: Michael Swift, Shivaram Venkataraman, and Kassem Fawaz. Michael and Shivaram were also members of my preliminary exam committee. They posed questions that prompted me to think deeply about my work on sharing and scheduling for persistent memory. I have always enjoyed my conversations with Michael and Shivaram. I would also like to thank Kassem for his willingness to serve on my committee and for his constructive feedback on my dissertation. Thank you, Michael, Shivaram, and Kassem.

Apart from the above, I had the privilege of working with outstanding researchers at UW-Madison and the Microsoft Jim Gray Systems Lab, including Mark Hill, Jignesh Patel, Xiangyao Yu, Kwanghyun Park, Rathijit Sen, and Brian Kroth. I took Mark Hill's architecture class. And many of his research tips, such as always attempting to do a taxonomy, inspired me greatly. That will be a life-long lesson for my future research. Jignesh is an energetic and inspiring professor. I took his advanced databases class, and after that I decided to do a lot of inter-discipline research across systems and databases. I worked on several projects with Xiangyao Yu. He is extremely smart, and always can come up with many neat designs. It has been a pleasure to work with him. Finally, Kwanghyun, Rathijit, and Brian from Gray Systems Lab have been very supportive to my Ph.D. research. I have learned a lot from working with them on a variety of projects. Working with many of these outstanding researchers was extremely beneficial to me during my Ph.D. studies.

I am grateful for having worked with a group of smart colleagues at school, including Vinay Banakar, Youmin Chen, Yifan Dai, Guanzhou Hu, Surabhi Gupta, Jing Liu, Anthony Rebello, Kaiwei Tu, Chenhao Ye, Sudarsun Kannan, Yuvraj Patel, Aishwarya Ganesan, Ram Alagappan, Jun He, Zev Weiss, Tyler Harter, Lanyue Lu, Yupu Zhang and more. I'd like to thank Lanyue Lu and Yupu Zhang for directing me to Remzi and Andrea's group while I was still applying to graduate schools. I am grateful to Jun He, with whom I worked during my first two years of graduate school. We work together to build a search engine from the ground up. Working with Jun has been extremely beneficial to my future endeavors. I'd like to thank Guanzhou Hu for his assistance with the Orthus project. I'd like to thank Kaiwei Tu for his help with the NyxCache project. I recall one week when we spent almost all of our time together in the computer science building's seventh floor conference room finishing a large amount of implementation and experiments. I'd like to thank Vinay Banakar for many discussions on PM-related topics, and our discussions have inspired many new ideas that we plan to pursue in the future. I'd also like to thank Ram Alagappan and Yuvraj Patel for their collaborations. Ram's logical thinking and writing abilities impress me when we work on the Orthus project. It was very inspiring to discuss with Yuvraj when we were working on the NyxCache project. Finally, I thank Jing Liu for many enjoyable and rewarding discussions. She is extremely intelligent and enthusiastic about a wide range of research topics.

I'd also like to thank my friends in Madison: Zhenmei Shi, Zifan Liu, Yuqun Zhou, Haoru Song, Wanting Wei, and many others. The many basketball games we played together and the gatherings we had helped me get through the difficult COVID time.

This Ph.D. would not have been possible without the support of my parents and family. My parents are very concerned about their only son's education. They did everything they could to help me get better education and living conditions. Unfortunately, my father passed away two months before my Ph.D. oral defense. But I believe he will be proud of his son's achievement. This Ph.D. is dedicated to him. I am also grateful to my family for their support while I was in graduate school. The last six years have been difficult for me due to health issues, COVID, and the inability to return to China for an extended period of time. I especially appreciate their assistance to my mother and father when I am unable to return home.

Finally, I cannot express how grateful I am to Zhihan, my wife. I am lucky to have a single person who is my true love, a wonderful life partner, and an inspiring colleague. I consider myself extremely fortunate to have met Zhihan in graduate school and fallen in love with her. Zhihan has been a tremendous emotional support throughout my Ph.D. journey. And we raise three adorable pets: Dongpo the rabiit and two Yorkies: Nori and Solo. Zhihan is also an excellent researcher who works on databases. We talk about a variety of projects all the time, and she is a co-author on my most proud Orthus paper. Zhihan is a wonderful life partner from whom I have learned a lot. She has cooked me countless delicious meals and taught me how to identify my life goal and stay calm in the face of chaos. I'm looking forward to working and living with her for many years to come. I love you, Zhihan.

Contents

Acknowledgments **ii**

Contents **vi**

Abstract **x**

1 Introduction **1**

1.1 Understanding Persistent Memory (PM) Devices Characteristics 4

1.2 Evolving Caching for PM Hierarchies 6

1.3 Evolving Sharing Mechanisms for PM 9

1.4 Contributions and Highlights 12

1.5 Overview 13

2 Persistent Memory Background **15**

3 Understanding PM Devices Characteristics **20**

3.1 The Unwritten Contract of Optane SSD 21

3.1.1 Access with Low Request Scale 22

3.1.2 Random Access is OK 27

3.1.3 Avoid Crowded Accesses 30

3.1.4 Control Overall Load 31

3.1.5 Avoid Tiny Accesses 32
3.1.6 Issue 4KB Aligned Requests 33
3.1.7 Forget Garbage Collection 34
3.2 Discussion and Implications From the Contract 37
3.2.1 Flash vs. Optane SSD 37
3.2.2 Implications From the Contract 38
3.3 Conclusions 41

4 Evolving Caching for PM Hierarchies 42
4.1 Background and Motivation 43
4.1.1 Managing the Storage Hierarchy 44
4.1.2 Hardware Storage Trends 45
4.2 Characterizing Caching in Traditional and Modern Storage Hierarchies 47
4.2.1 Modeling Caching Performance 48
4.2.2 Evaluation with Optane DC PM and Optane SSD . . 52
4.3 Non-Hierarchical Caching 55
4.3.1 Formal Definitions 57
4.3.2 Architecture 58
4.3.3 Cache Scheduler Algorithm 60
4.4 Implementation 63
4.5 Evaluation 64
4.5.1 Orthus-CAS 65
4.5.2 Orthus-KV: Static Workloads 69
4.5.3 Orthus-KV: Dynamic Workloads 72
4.5.4 Comparisons with Prior Approaches 76
4.6 Conclusions 78

5 Evolving Sharing Mechanisms for PM 79
5.1 Background and Motivation 80
5.1.1 Sharing Policies for Multi-Tenant Caches 80

5.1.2 Challenges of PM Cache Sharing 84
5.2 NyxCache Design 89
5.2.1 Architecture 89
5.2.2 Design goals 91
5.2.3 Nyx Mechanisms 91
5.2.4 Nyx Sharing Policies 95
5.2.5 Cache Instances: PM-Optimized Pelikan 101
5.2.6 Nyx Parameter Values 102
5.3 Evaluation 103
5.3.1 Mechanisms Overhead 104
5.3.2 Resource Limiting 105
5.3.3 QoS-Aware 107
5.3.4 Fair Slowdown 110
5.3.5 Proportional Resource Allocation 111
5.3.6 Realistic Traces 115
5.3.7 Parameters Sensitivity Analysis 116
5.4 Discussion 118
5.5 Conclusions 119

6 Related Work **120**
6.1 Characterizing Modern PM Devices 120
6.2 Evolving Caching for PM Hierarchies 122
6.3 Evolving Sharing Mechanisms for PM 126

7 Conclusions **128**
7.1 Summary 128
7.1.1 Characterizing Modern PM Devices 129
7.1.2 Evolving Caching for PM Hierarchies 130
7.1.3 Evolving Sharing Mechanisms for PM 130
7.2 Lessons Learned 131

7.2.1 Understanding Device Characteristics is Critical for Storage Research 131
7.2.2 It is Helpful to Think Like a Novice at Some Point . 133
7.2.3 It is Important to Separate Mechanisms and Policies in System Research 134
7.3 Future Work 135
7.3.1 Energy Efficiency Characterization of PM Devices . 135
7.3.2 Non-hierarchical Caching in the world of Memory/Storage Disaggregation 136
7.3.3 Sharing with Better Hardware/Software Co-design 137
7.3.4 Sharing of PM and DRAM pool in Disaggregated Memory Setup 137
7.4 Discussion: Demise of Intel Optane Memory Business . . . 138
7.4.1 Implications for This Thesis 138
7.4.2 Implications of Our Thesis for Future PM Device Development 139
7.5 Closing Words 140

Bibliography **142**

Abstract

Data has been critical to human society, and data storage systems that hold the data of every human being are playing an important role in people's modern lives. These storage systems serve people's information in a variety of places, from mobile devices and laptops to large-scale data centers and the cloud. People rely on these storage systems for critical applications such as e-commerce, social media, health care, and artificial intelligence. In the modern world, people interact with data storage systems almost every moment. As a result, there is a high demand for data storage systems that are reliable, large, performant, and cost-effective.

To meet these demands, the system community has been working on high-performance, low-cost storage devices as well as software system stacks to manage them. For decades, numerous storage devices such as tape, DRAM, HDD, and SSD have been invented. Meanwhile, system stacks to manage these devices have evolved, ranging from primitive data structures (e.g., B-tree) to complex file systems and distributed storage systems. This dissertation focuses on Persistent Memory (PM), a new development in the storage device landscape. We investigate how to evolve the system stack for new PM devices.

In the first part of this thesis, we characterize a popular PM-based device – the Intel Optane SSD. We investigate the performance of Optane SSDs in response to a variety of micro-experiments. We formalize an "unwritten contract", which includes rules that are essential for Optane SSD users to achieve optimal performance. We also use carefully designed experiments to reveal the internals of Optane SSDs. We reveal Optane SSD's internal parallelism, read-write scheduling mechanisms, and so on. These internals provide insights about the unwritten contract. Finally, we discuss implications of our device characterization.

In the second part, we investigate how classic caching performs on modern storage hierarchies, with new PM devices filling the gap between DRAM and SSD. We compare DRAM/PM/Low-latency SSDs/Flash SSDs quantitatively. Our analysis shows that modern hierarchies have a much smaller performance difference between neighboring layers than traditional hierarchies. We then examine caching on modern hierarchies. Caching manages data across two layers: a cache layer (faster but smaller) and a capacity layer (slower but larger). We find that classic caching, which directs as many accesses to the cache layer as possible (referred as the principle of maximizing hit rates), cannot effectively utilize the significant available performance in today's capacity layer (e.g., PM). Thus, we introduce three improvements (read around, admission rejection, and feedback-based offloading) that evolve classic caching for PM hierarchies.

In the final part of this thesis, we study how unique characteristics of PM influence the effectiveness of existing sharing mechanisms. We focus on a detailed setup of multi-tenant in-memory key-value caches. We first summarize the basic mechanisms (for example, resource usage accounting, interference analysis, etc.) used to achieve diverse sharing goals. We then investigate these mechanisms on PM. Our analysis shows that existing sharing mechanisms designed for DRAM/block devices do not readily translate to PM due to PM's unique characteristics such as 256B access granularity and severe and unfair interference between reads and writes. To address this issue, we present Nyx, a PM access regulation framework that is optimized for today's PM without special hardware support. Nyx introduces new software sharing mechanisms for PM that can be be used to easily and efficiently support sharing policies such as resource limiting, QoS, fair slowdown, and proportional sharing.

1 Introduction

Data storage systems have become indispensable in modern society. File systems [24], which manage data on mobile devices [44], computers [225], and data centers [19], enable us to store and retrieve information daily. Every second, databases [201], such as those used in e-commerce [4, 119] and social media [68, 91, 239], process billions of queries to critical data in the lives of individuals. Meanwhile, data-intensive artificial intelligence applications [9, 156] will continue to necessitate large-scale distributed storage systems [8, 49, 74]. In the era of cloud computing, the demand for large, reliable, efficient, and cost-effective data storage systems continues to grow [5, 20, 26].

To meet these demands, the system community has been developing high-performance, low-cost storage devices, as well as the software system stack to manage them [21]. For primary storage/memory, we've seen various versions (DDR1-5) of Dynamic Random Access Memory (DRAM) [23] and High Bandwidth Memory [29] over the last decade. For secondary storage, magnetic tapes [50] were used for the very first computers [75], while Hard Disk Drives (HDD) [28], and Solid-State Drives (SSD) [73] are excellent examples of more recent devices. To exploit each type of device, a large number of systems/techniques have been developed: from the design of primitive data structures (e.g., B+ tree [11], Log-structured Merge-tree [48]) to complicated data management systems in a single machine (e.g., file system [170, 180], databases [125, 201]), and distributed systems (e.g., RamCloud [193], Google File System [136]). The system stack has evolved in response to different characteristics of the storage devices.

Persistent Memory (PM) [63] is one of the most recent developments in the storage device landscape. PM provides memory-like byte-addressable accesses while ensuring data persistence in the event of a failure. PM is expected to achieve DRAM-like speed, while having a larger capacity and less cost compared to DRAM [1, 63, 77, 169]. There has been a lot of interest in using PM for databases [129], big data processing [137], cloud computing [106], and artificial intelligence applications [130].

However, due to the unique characteristics of PM, existing systems must be rethought or redesigned for PM. PM is different to DRAM. First, PM has persistence, so systems that require persistence from PM must manage crash consistency. Second, unlike DRAM, PM exhibits highly asymmetric read vs. write performance, and is especially efficient for multiples of 256B accesses [238]. Previous DRAM-based systems must be reevaluated in light of these significant DRAM and PM differences. PM also differs from SSD. It is byte-addressable and has unprecedented sub-microsecond access latency. Traditional SSD/HDD block interfaces are hence inefficient, and expensive software stacks (e.g., the OS block layer) are no longer appropriate for PM [90]. To fully realize the performance potential of the new PM devices, a fundamental question arises: **how should the system stack evolve for PM devices?**

In this dissertation, we attempt to address this question from three perspectives: i) understand the performance characteristics of PM devices, ii) rethink caching on PM hierarchies, and iii) rethink sharing mechanisms for PM.

We begin by characterizing and revealing the internals of recently commercialized PM devices. We characterize the performance of the Intel Optane SSD [40], by examining its response to various access patterns.

Based on our performance studies, we summarize rules that Optane SSD users must follow in order to achieve optimal performance; we refer to these rules as the “unwritten contract” of Optane SSD. The device characterization serves as a foundation for evolving the system stack for PM.

Next, we analyze classic caching [14] on modern storage hierarchies [103], with new PM layers [37] filling the performance gap between DRAM and SSD. Consider a system with two storage layers: a (fast, expensive, small) performance layer like DRAM and a (relatively slow, cheap, large) capacity layer like PM. Caching manages data placements between these two layers. We find that: classic caching strives to direct as many accesses as possible to the cache layer (known as the maximizing cache hit rates principle); as a result, caching does not use capacity layer performance even when the cache device is fully saturated. And today, capacity layer performance, such as PM performance, can account for a significant portion of total performance out of the hierarchy. After understanding why the principle does not work well, we introduce three ways for evolving classic caching on PM hierarchies: i) read around, ii) admission rejection, and iii) feedback-based offloading.

Finally, we study how the unique characteristics of PM influence the effectiveness of existing sharing mechanisms. We aim to enable PM sharing across multiple tenants while achieving performance isolation, fairness, or other goals. We find that both DRAM and SSD/HDD sharing mechanisms [35, 43] do not translate into effective PM sharing approaches. We perform a detailed analysis of each mechanism’s problems and propose new mechanisms that better enforce sharing goals on PM.

We believe that our studies will prompt and stimulate the system community to consider how to evolve other aspects of the system stack for PM, and thus significantly reshape future PM systems used in databases, cloud computing or data processing systems that support our daily lives.

1.1 Understanding Persistent Memory (PM) Devices Characteristics

We first seek to understand the performance characteristics of PM devices. Researchers have been developing various types of PM technologies for decades. Only recently, with the 3D XPoint Memory technique from Micron and Intel [1, 141], has the system community gained access to widely-available PM devices. Intel made storage devices based on 3D XPoint Memory available in various form factors, including Optane SSD [39] (a block device), and Optane DC PM [37] (byte-addressable memory DIMMs). The first part of the dissertation focuses on the Intel Optane SSD, which is the most cost-effective of the PM devices and the only widely available option at the time of our study.

Optane SSD offers numerous opportunities for applications. For example, Intel's Memory Direct Technology (IMDT) [36] enables the use of Optane SSD as a DRAM alternative. Use cases of IMDT/Optane SSD include Memcached [31], Redis [32], and Spark [33]. Evaluations of those scenarios demonstrate the potential role of Optane SSD as a cost-effective alternative of DRAM. Optane SSDs are also deployed to support key workloads in Facebook [129, 130], both as a caching layer between DRAM and Flash SSD for RocksDB and for crucial workloads such as machine learning. According to Eisenman *et al.* [130], Facebook stores embedding of trained neural networks on Optane SSDs.

Using new technology effectively requires understanding its performance and reliability characteristics. For traditional devices, such as HDDs and Flash-based SSDs, their characteristics are well known [101, 102, 123, 124, 143, 208, 237]. However, for new devices like the Optane SSD, much remains unclear, and is thus the focus of our work. We conduct fine-grained experiments to characterize the Optane SSDs and summarize the "unwritten contract" that Optane users must adhere to for optimal performance.

In terms of immediate performance, we summarize six rules that are critical for Optane SSD users. First, to obtain low latency, users should issue small requests ($\geqslant 4$ KB) and keep a small number of outstanding IOs (**Access with Low Request Scale** rule). Second, different from HDD/Flash SSDs, Optane SSD clients should not consider sequential workloads special (**Random Access is OK** rule). Third, to avoid contention among requests, clients should not issue parallel accesses to a single chunk (4KB) (**Avoid Crowded Accesses** rule). Fourth, to achieve optimal latency, the user needs to control the overall load of both reads and writes (**Control Overall Load** rule). Fifth, to exploit the bandwidth of Optane SSD, clients should never issue requests less than 4KB (**Avoid Tiny Accesses** rule). Sixth, to get the best latency, requests issued to Optane SSD should align to eight sectors (**Issue 4KB Aligned Requests** rule). Finally, when serving sustained workloads, there is no cost of garbage collection in Optane SSD (**Forget Garbage Collection** rule).

In order to present insights regarding these rules, we use carefully crafted micro-experiments to reveal the internal of Optane SSDs. We experimentally reveal the amount of internal parallelism, read-write scheduling mechanisms, block alignment granularity, and mapping policies from logical-block address (LBA) to physical-block address (PBA) in the Optane SSD. Overall, we show that the Optane SSD unwritten contract is relevant to features of 3D XPoint memory and Intel Optane SSD's controller/interconnect design.

The unwritten contract provides numerous implications for systems and applications. For example, according to the Access with Low Request Scale rule and the Control Overall Load rule, the design of heterogeneous storage systems including Optane SSD and Flash SSD must be carefully considered. Moreover, many rules (e.g., Random Access is OK) present opportunities and new challenges to external data structure design for

Optane SSD. Finally, the Random Access is OK rule may enable new applications on Optane SSD that don't work well on existing technologies. In general, our device characterization study provides a foundation for our subsequent system stack evolution for PM, such as the following caching work.

1.2 Evolving Caching for PM Hierarchies

Our performance studies show that the modern storage landscape is complicated by new PM devices, and we believe that existing systems need to be reevaluated/rethought. The second part of the dissertation focuses on the change in storage hierarchy and the associated caching approaches. We demonstrate the limitations of classic caching approaches on modern storage hierarchies with PM and evolve caching for these hierarchies.

The notion of a *hierarchy* (i.e., a *memory hierarchy* or *storage hierarchy*) has long been central to computer system design. Indeed, assumptions about the hierarchy and its fundamental nature are found throughout widely used textbooks [103, 144, 212]: "Since fast memory is expensive, a memory hierarchy is organized into several levels – each smaller, faster, and more expensive per byte than the next lower level, which is farther from the processor.[144]"

To cope with the nature of the hierarchy, systems usually employ two strategies: *caching*[14, 191] and *tiering*[16, 138, 232]. Consider a system with two storage layers: a (fast, expensive, small) performance layer and a (slow, cheap, large) capacity layer. With caching, all data resides in the capacity layer, and copies of hot data items are placed, via cache replacement algorithms, in the performance layer. Tiering also places hot items in the performance layer; however, unlike caching, tiering migrates data (instead of copying) on longer time scales. With a high-enough fraction of requests going to the fast layer, the overall performance approaches the peak performance of the fast layer. Consequently, classic caching and tiering strive to ensure that most accesses hit the performance layer.

While this conventional wisdom of maximizing hit rates may remain true for traditional hierarchies (e.g., CPU caches and DRAM, or DRAM and HDDs), rapid changes in storage devices have complicated this narrative within the modern storage hierarchy. Specifically, the advent of PM DIMMs [81, 158, 202] and low-latency SSDs [40, 56, 71] bridges the performance gap between DRAM and SSD; they introduce neighboring layers with much closer and (sometimes) overlapping performance in today's hierarchies. Thus, it is essential to rethink how such devices must be managed in the storage hierarchy.

To understand this issue better, consider a two-level hierarchy with PM devices (50GB/s total bandwidth) as the capacity layer, and traditional DRAM (100GB/s total bandwidth) as the performance layer. As we will show (§4.5), when the workload is light, DRAM latency is an order of magnitude lower than PM, and thus the traditional caching/tiering arrangement works well. However, in other situations (namely, when the workload has a high concurrency), even assuming a perfect 100% hit rate, classic caching and tiering can only utilize DRAM bandwidth while leaving a significant mount of PM bandwidth idle. To maximize performance in modern PM hierarchies, a different approach is needed.

To address this problem, we introduce *non-hierarchical caching* (NHC), a new approach to caching for modern storage hierarchies. NHC delivers maximal performance from modern devices despite complex device characteristics and changing workloads. The key insight of NHC is that when classic caching would send more requests to the cache/performance device than is useful, some of that excess load can be dynamically moved to the capacity device. NHC improves upon classic caching in three ways: i) read around, ii) admission rejection, and iii) feedback-based offloading. First, NHC enables offloading of read hits to capacity devices. NHC monitors cache performance and adapts the requests sent to each device, thereby delivering additional useful performance from the capacity device. Second, NHC turns off the data admission to cache devices when this movement does not improve overall cache performance. Finally, NHC determines the amount of offloading at runtime by checking cache performance feedback when NHC fine-tunes the offloading ratios. While the idea of redirecting excess load to devices lower in the hierarchy applies to both caching and tiering, we focus on caching.

Previous work has addressed some of the limitations of caching [80, 160], offloading excess writes from SSDs to underlying hard drives. However, as we show (§4.5.4), they have two critical limits: they do not redirect accesses to items present in the cache (hits), and they do not adapt to changing workloads and concurrency levels (which is critical for modern devices).

We implement NHC in two systems: Orthus-CAS, a generic block-layer caching kernel module [108], and Orthus-KV, a user-level caching layer for an LSM-tree key-value store [168]. Under light load, Orthus implementations behave like classic caching; in other situations, they offload excess load at the caching layer to the capacity layer, improving performance. Through rigorous evaluations, we show that Orthus implementations greatly improve performance (up to 2$\times$) on various real devices (such as Optane DC PM, Optane SSD, Flash SSD) and other simulated ones for a range of workloads (YCSB [117] and ZippyDB [107]). We show NHC is robust to dynamic workloads, quickly adapting to load and locality changes. Finally, we compare NHC against prior caching strategies and demonstrate its advantages. Overall, the non-hierarchical approach extracts high performance from modern storage hierarchies.

1.3 Evolving Sharing Mechanisms for PM

So far, we primarily focused on a single application or tenant's point of view of using PM. Because PM is large and cheap, there is a great deal of interest in using it in a shared environment. As a result, a question arises: how can we share PM across multiple tenants? In the final part of the thesis, we address this question. We examine prior sharing mechanisms, designed for DRAM/block devices, on PM, expose their fundamental limitations, and then propose new mechanisms to address PM sharing challenges. We concentrate on a detailed setup of multi-tenant in-memory key-value caches.

Memory-based look-aside key-value caches (e.g., memcached [52]) are a critical component of many systems and applications [10, 17, 88, 239]. To improve utilization and simplify management, multiple cache instances are often consolidated onto a single multi-tenant server. For example, Facebook [187] and Twitter [239] each maintain hundreds of dedicated cache servers that host thousands of cache instances. However, multi-tenant servers have the added challenge of ensuring that each client cache meets its performance goals; a range of production and research in-memory multi-tenant caches currently provide different sharing policies, such as enforcing a limit on the used memory capacity and bandwidth [27], guaranteeing a level of quality-of-service (QoS) [62], and allocating resources proportionately [199].

PM, such as that provided by Intel's Optane DC PM [37], is emerging as an appealing building block for these caches, due to PM's large capacity, low cost per byte, and comparable performance to DRAM. However, PM performance differs from DRAM and Flash in a number of ways that reduce the effectiveness of current multi-tenant caches for other devices [116, 214]. In particular, unlike DRAM, Optane DC PM exhibits highly asymmetric read vs. write performance (for a single DC PM, max read bandwidth is 6.6GB/s whereas max write bandwidth is

2.3GB/s) [151], severe and unfair interference between reads and writes (writing at 1GB/s can cause the same throughput and P99 latency slowdown to a co-running read workload as reading at 8GB/s) [189], and especially efficient access for multiples of 256B [238].

Unfortunately, existing multi-tenant DRAM and storage caching techniques do not readily translate to PM. Some approaches focus exclusively on capacity allocation across clients [116, 199, 214]; capacity allocation is necessary but not sufficient for PM sharing because the rate of requests to PM must also be regulated. Host-level request regulation has been explored extensively for Flash devices using block-layer I/O scheduling [197, 210], but these software overheads are prohibitive given 100ns PM accesses [90]. Device-level request scheduling assumes special hardware that PM lacks [186, 217, 245, 247]. Finally, coarse-grain request throttling underpins the vast majority of DRAM bandwidth allocation techniques; however, these approaches assume both hardware counters and performance characteristics that do not hold for PM (e.g., bandwidth is an accurate estimate of utilization).

In the last part of the thesis, we introduce NyxCache (Nyx), a standalone lightweight and flexible PM access regulation framework for multi-tenant key-value caches that is optimized for today's PM without special hardware support. Given a PM server and a sharing policy (e.g., QoS), cache instances are admitted and assigned space using existing load admission [120, 121, 179] and capacity allocation [116, 199, 214] techniques. At runtime, Nyx monitors information (e.g., PM resource utilization) of caches, regulates the rate at which each cache is allowed to access PM, and thus enforces the sharing policy's performance goals. Nyx works with any in-memory key-value store that adheres to the memcached interface [52]; the current implementation includes a PM-optimized version of Twitter's Pelikan [61] that can improve single-cache performance by more than 50% for get-heavy workloads and 3× for write-heavy workloads.

Nyx's central contribution is a set of software mechanisms designed for PM to extract the information required to flexibly enforce popular sharing policies.

Nyx provides new mechanisms to efficiently i) regulate PM accesses, ii) obtain a client's PM resource usage, iii) analyze inter-client interferences, and has two particularly useful and novel mechanisms for PM. First, Nyx efficiently estimates not only the total PM DIMM utilization (building on pioneering work in this space [189]), but also the PM utilization caused by each cache instance, as is needed for sharing policies; estimating PM utilization is challenging because the number of transferred bytes is not an accurate proxy of PM utilization, unlike on DRAM. Second, Nyx can determine which cache instance most interferes with another cache instance; in PM-based systems, these interactions are difficult to identify because a harmed client may be impacted more by a low-bandwidth client than a high-bandwidth client, unlike DRAM. Both of these mechanisms accurately account for the CPU cache prefetching that is essential for high performance on PM. These new mechanisms enable Nyx to easily and efficiently support sharing policies such as resource limiting, QoS, fair slowdown, and proportional sharing.

The sharing policies provided by Nyx are powerful. Nyx can accurately limit the PM utilization of each cache (similar to Google Cloud's memcache [27]), whereas an approach that measures only bandwidth cannot. Nyx can provide QoS guarantees to latency-critical caches while providing higher throughput (up to 6$\times$) to best-effort caches that are not interfering. Nyx can provide proportional resource allocation while redistributing idle PM utilization to clients that will not inadvertently slowdown others. Finally, as shown for real large-scale cache traces from Twitter, Nyx can isolate clients from write spikes and ensure that important caches are not slowed down by increased best-effort traffic.

1.4 Contributions and Highlights

We list the main contributions of this dissertation.

Understanding PM Devices Performance Characteristics.

- We provide the first comprehensive performance analysis of a popular PM-based device: the Intel Optane SSD.
- We formalize the "unwritten contract" that Optane SSD users must follow in order to achieve the best performance out of Intel Optane SSDs. We also present the impact when violating each rule.
- We design carefully-crafted experiments to reveal the internals of Optane SSD. Our study provides insights into each contract rule and serves as a knowledge base for future research.
- We provide implications from the unwritten contract for i) developing applications on Optane SSD, and ii) managing storage hierarchies with Optane SSD.

Evolving Caching on PM Hierarchies.

- We present the first quantitative comparison of device pairs in modern storage hierarchies, such as DRAM/PM/Low-latency SSD/Flash. Our analysis reveals a significant change in modern hierarchies: the performance difference between neighboring layers is now much smaller than in traditional hierarchies.
- We investigate caching on modern hierarchies using both performance modeling and measurements. The investigation shows that the decades-old caching principle of maximizing hit rates is no longer sufficient.

- We design and build Non-Hierarchical Caching (NHC), which evolves caching for modern hierarchies. Non-hierarchical caching is a generic approach; any classic caching implementation can be augmented by incorporating the read around and admission rejection mechanisms in NHC.

Evolving Sharing Mechanisms for PM.

- We provide a summary of basic mechanisms that are commonly used to achieve diverse sharing goals. This summary can help future studies on PM and other device sharing.
- We demonstrate that prior DRAM or storage device-intended approaches for access regulation, resource-usage estimation, and interference analysis fail to work on PM due to PM's unique performance characteristics.
- We introduce Nyx, a sharing framework which enables these sharing mechanisms on PM. We devise software mechanisms that can be immediately applied to today's PM devices without special hardware assumptions.
- We revise popular sharing policies (i.e., QoS aware, Resource Limiting, Proportional Sharing, Fair Slowdown) upon PM based on Nyx mechanisms.

1.5 Overview

We briefly describe the contents of the chapters of this dissertation.

- **Persistent Memory Background.** In Chapter 2, we provide relevant PM background. We provide high-level overview of PM techniques and compare PM to DRAM and Flash. Then, we introduce two commercially available PM devices today: the Intel Optane SSD and Intel Optane DC PM.

- **Understanding PM Devices Characteristics.** In Chapter 3, we first present our experimental analysis of the Intel Optane SSD. We then summarize the unwritten contract of the Optane SSD. We also discuss implications from the contract.

- **Evolving Caching for PM Hierarchies.** Chapter 4 first describes how PM devices have strong implications on caching in modern storage hierarchies. Then, in the second part, we present non-hierarchical caching, a method to augment classic caching implementations for PM hierarchies. We describe the design, implementation, and our evaluation.

- **Evolving Sharing Mechanisms for PM.** In Chapter 5, we explore how to share PM across multiple tenants. We investigate existing DRAM/block devices sharing approaches' problems on PM. We then describe how Nyx supports PM access regulation, resource usage accounting, and cross-tenant interference analysis on PM.

- **Related Work.** In Chapter 6, we first describe previous work on block device characterization, which is relevant to our investigation of Optane SSDs. We then discuss prior techniques that have been developed to effectively manage classic or modern storage hierarchies. Finally, we discuss work related to multi-tenant PM sharing.

- **Conclusions and Future Work.** In Chapter 7, we summarize the dissertation and the lessons we learned while working on it. Finally, we discuss how the work presented in this dissertation can be expanded upon, as well as the future directions our work suggests.

2 Persistent Memory Background

In this chapter, we provide necessary background on PM techniques. In addition, we highlight key differences between PM and existing popular DRAM and Flash Devices [25]. Finally, we introduce two types of commercially available PM devices, which we will use throughout the remainder of the dissertation.

Persistent Memory. Persistent memory refers to a class of technologies which provide "byte addressable" storage that survives power outages [63]. PM distinguishes itself through two defining characteristics: memory-like access and persistence. It should be compatible with microprocessor memory instructions (such as load and store). It can also perform remote direct memory access (RDMA) [67] actions like RDMA read and RDMA write. In addition, PM should be able to retain program data or state that exists outside of the fault zone of the process that generated it (i.e., persistence). The capabilities of persistent memory typically go beyond the non-volatility of stored data. For example, the loss of metadata such as page table entries that translate virtual addresses to physical addresses may result in non-persistent but durable content. Hence, most PM technologies include some basic file systems (similar to typical storage devices) for associating names or identifiers with stored data.

In the past decade, numerous PM prototypes have been developed, including spin-transfer torque, phase change, and resistive memo-

Device Type	**Latency**	**Typical Single-Device Capacity**	**Cost** ($/GB)
DRAM	80ns	16, 32, 64 GB	~7
Optane DC PM	300ns	128, 256, 512 GB	~5
Optane SSD	10us	400, 800, 1600 GB	1
Flash SSD	80us	1-10s TB	0.3

Table 2.1: **Comparing PM Devices to DRAM and Flash.** *This table compares PM-based SSD (Intel Optane SSD [39]), and PM DIMMs (Intel Optane DC PM [37]) to DRAM and Flash.*

ries [81, 105, 158, 202]. For example, phase change memory [202] is based on a Chalcogenide glass that can be programmed by varying electrical inputs to the cell. Resistive memory [233] is based on metal oxide, and spin-transfer torque memory [105] stores data using electron spin. Many of these PM techniques have features in common, such as relatively slower accesses than DRAM and asymmetric read-write latencies. Meanwhile, PM's write endurance will be limited. With the recent introduction of Micron and Intel's commercially available 3D XPoint Memory technique [1, 141], the system community can now evaluate PM's true capabilities, design systems for real devices, and deploy them in a variety of scenarios.

PM vs. DRAM and Flash. PM is different from DRAM [23] or Flash SSDs [25]. DRAM is a type of random-access semiconductor memory that stores bits in memory cells composed of capacitors and transistors. DRAM is commonly constructed using metal-oxide-semiconductor technology. DRAM is typically packaged in the form of memory DIMMs that connect to memory buses that are linked to CPUs. Flash memory (or, more precisely, NAND-based flash memory) is a type of floating-gate memory invented in the 1980s by Toshiba [25]. Flash memory can be erased and reprogrammed electrically. To write to a flash page, for example, we must first erase a larger chunk of the flash block. Flash memory is usually found in solid-state drives (SSD). SSDs are connected to either the SATA [72] or PCIe [60] bus.

PM is persistent, larger and less expensive than DRAM (as shown in Table 2.1). For example, a single server can be outfitted with up to 12TB of Intel Optane DC PM [37] (one version of PM device), making PM appealing for developing memory-intensive applications. Meanwhile, because PM is less expensive than DRAM (Table 2.1), large memory can now be cost-effective. In terms of performance, current PM devices are three to six times slower than DRAM [238]. Meanwhile, PM has asymmetric read and write performance, as opposed to DRAM. As we will see more in our following studies, PM has unique performance characteristics compared to DRAM; in other words, PM is more than just a slower DRAM.

Compared to Flash SSDs, PM is byte-addressable, can be directly connected to the memory bus, and supports hardware load/store accesses. These features are extremely beneficial to application developers. PM byte-addressability, for example, avoids the serialization and deserialization overheads required by SSDs and other block devices. In terms of performance, when compared to Flash SSDs, PM delivers unrivaled performance (as we will show, for example, sub-microsecond access latency, supreme random access performance, and high maximum bandwidth). Because of these distinguishing characteristics, existing applications designed for SSDs must be thoroughly rethought for PM.

Commercially-available PM Devices. Based on 3D XPoint Memory [1], Intel made available storage devices in a variety of form factors. Currently, the most popular options are Intel Optane SSD [39] and Intel Optane DC PM [37].

The Intel Optane SSD is the first widely available PM-based hardware, and it is a low-latency block device built on 3D XPoint Memory. It connects to the PCIe 3.0 bus, just like a Flash-based SSD, and Optane SSD users access it through the traditional block interface. Its first-generation

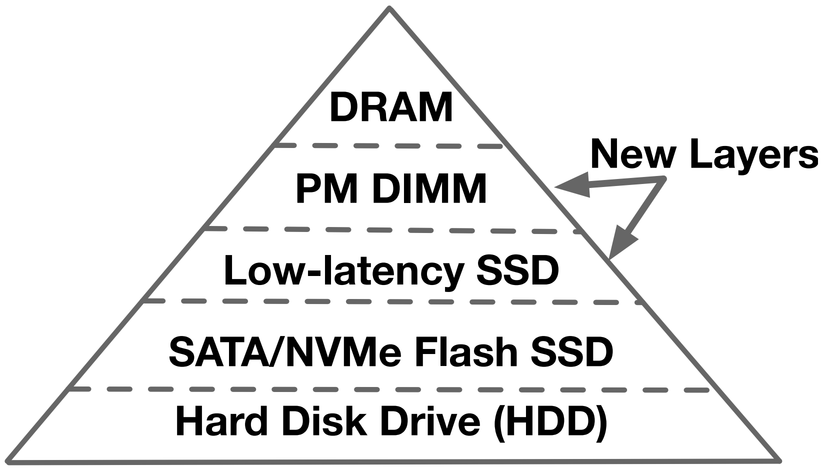


Figure 2.1: **Modern Storage Hierarchies.** *This figure contrasts modern and traditional storage hierarchies. We have new PM DIMM (Optane DC PM or other future PM devices connected to memory buses) and low-latency SSDs (typically based on some form of PM techniques) layers in modern storage hierarchies that bridge the performance gap between DRAM and SATA/NVMe Flash SSD.*

bandwidth is limited by PCIe 3.0 channel bandwidth limits; however, Intel claims that its bandwidth potential will increase with future fast PCIe versions. Because of the low latency of 3D XPoint Memory [1], Optane SSDs provide an order of magnitude lower access latency than Flash SSDs (Table 2.1). Due to the block interface of Optane SSDs, previous storage applications (built for SSDs/HDDs) can benefit from PM's high performance without requiring any changes to the implementation. On the other hand, Optane SSD device cannot benefit from direct load/store operations, all accesses to Optane SSD still require software OS or block layer interception overhead.

In contrast to Optane SSDs, Intel Optane DC PM DIMMs sit directly on the memory bus. Intel Optane DC PM can also be accessed via RDMA actions. The Optane DC PM supports *Memory* and *App Direct* modes [37]. The Memory mode uses DRAM as a transparent cache in front of PM without persistence, exposing DRAM and PM as a whole large memory to end users. The App Direct mode allows software to access PM and DRAM via normal `load` and `store` instructions with persistence. On Optane DC PM, data allocation, naming, and access are usually handled by a file system [234] or another management layer. Optane DC PM is accessible via fast network connect such as RDMA.

Optane SSD and Optane DC PM have piqued the interest of many people who seek to build large, performant, and cost-effective data storage systems. Optane SSD is not the typical PM by definition. However, due to its lower price compared to Optane DC PM (Table 2.1), no requirement for specific CPU versions, and ease of porting from existing systems, they remain popular today. Optane DC PM and Optane SSDs have evolved into two new layers in modern storage hierarchies. Optane DC PM devices, as shown in Figure 2.1, bridge the gap between DRAM and SATA/NVMe Flash SSDs.

3 Understanding PM Devices Characteristics

In this chapter, we characterize the performance of the new type of PM devices. At the time of the study, Intel Optane SSD was the only widely accessible PM option, and thus become the focus of our investigation.

In the first part of this chapter, we examine the performance of Optane SSDs in response to numerous fine-grained experiments. Based on our performance studies, we formalize an "unwritten contract" for the Optane SSD. The contract includes six rules that are critical for users of Optane SSD to achieve optimal immediate performance, as well as one rule for sustained workloads relating to garbage collection in Optane SSDs. Our analyses show that violating this contract can result in 11 times worse read latency and limited throughput (only 20% of peak bandwidth) regardless of parallelism. It is hence essential to follow the contract when using Optane SSDs.

Then, we use carefully crafted micro-experiments to reveal the internal of Optane SSDs in order to present insights regarding these rules. We experimentally reveal the internal parallelism, read-write scheduling mechanisms, block alignment granularity, and mapping policies from logical-block address (LBA) to physical-block address (PBA) in the Optane SSD. Overall, we demonstrate that the Optane SSD unwritten contract is relevant to features of 3D XPoint memory and Intel Optane SSD's

Rule	**Impact**	**Metric**	**Cause**
Access with Low Request Scale	11x	Latency	SSD Controller/Interconnect
Random Access is OK	7x (vs. Flash)	Latency & Throughput	3D XPoint & Controller
Avoid Crowded Accesses	4.6x	Latency & Throughput	SSD Controller/Interconnect
Control Overall Load	5x (vs. Flash)	Latency	3D XPoint Memory
Avoid Tiny Accesses	5x	Throughput	SSD Controller/Interconnect
Issue 4KB Aligned Requests	1.2x	Latency	SSD Controller/Interconnect
Forget Garbage Collection	15x (vs. Flash)	Sustained Throughput	3D XPoint & Controller

Table 3.1: **Performance Impact of Rule Violations.** *This table summarizes i) seven rules of the Optane SSD Unwritten Contract, ii) the maximum impact in terms of metric when violating each rule, and iii) Cause of the rules.*

controller/interconnect design.

Finally, we discuss the unwritten contract's implications and potential future works. We consider two types of audiences: first, system developers for Optane SSD; second, those who manage Optane SSD in storage hierarchies. This chapter is based on the paper *Towards an Unwritten Contract of Intel Optane SSD*, published in HotStorage 2019 [228].

In the following, we first present our Optane SSD characterization and related experiments to reveal Optane SSD internals in Section 3.1. Then, we discuss implications of the unwritten contract in Section 3.2. Finally, we summarize and conclude (Section 3.3).

3.1 The Unwritten Contract of Optane SSD

In this section, we define the rules of the unwritten contract. We offer experiments (available at https://github.com/sherlockwu/OptaneBench) to infer the rules. In Table 3.1, we summarize the impact and cause of each rule. We begin with six rules related to immediate performance, like latency and throughput, out of Optane SSDs. We then present rules for sustainable performance.

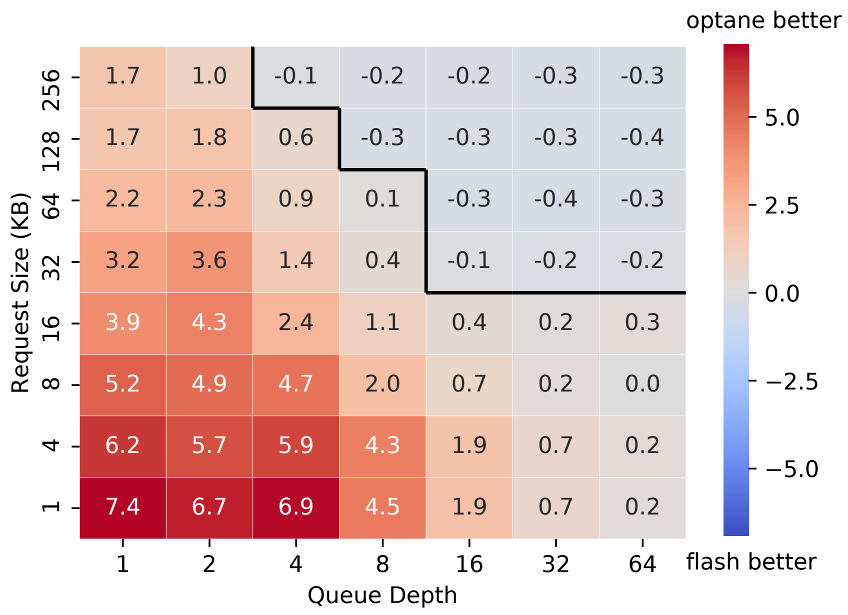
3.1.1 Access with Low Request Scale

The Optane SSD is based on 3D XPoint memory which is said to provide up to 1,000 times lower latency than NAND Flash [2]. Does 3D XPoint memory always lead to better performance for Optane SSD compared to Flash SSD? We answer this question with our first rule: to obtain low latency, Optane SSD users should issue small requests and maintain a small number of outstanding IOs. This rule is needed not only to extract low latency but also enough to exploit the full bandwidth of the Optane SSD.

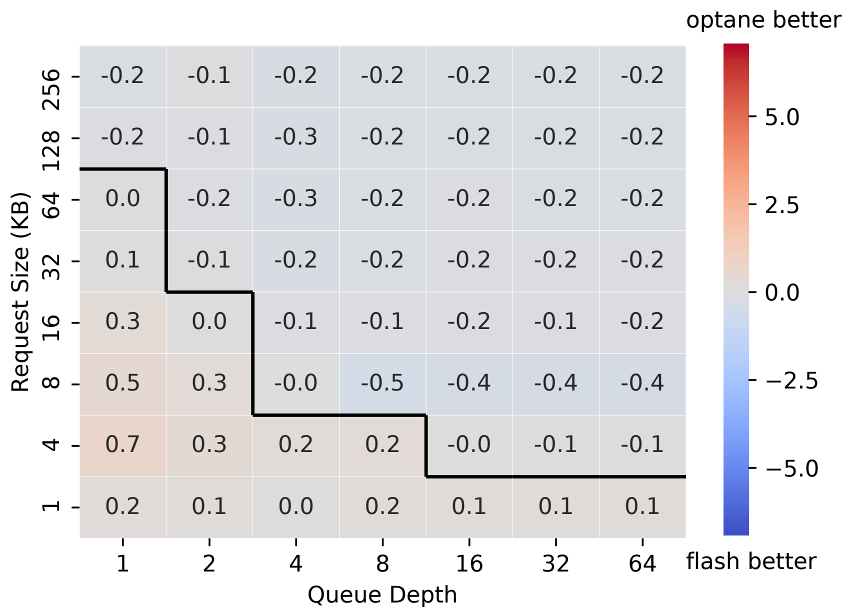
We uncovered this rule when quantitatively comparing Intel Optane SSD 905P (960GB) with a "high-end" Flash SSD: Samsung 970 Pro (1TB) [69]. From our experiments, we found that **Optane SSD does not show improvement for workloads with many outstanding requests**.

We compare Optane and Flash SSDs with random read-only and write-only workloads. Each workload has two variables: request size and queue depth (QD) (or number of in-flight I/Os). Figure 3.1 compares the two devices in terms of average access latency. The temperature T in each rectangle shows the scaled difference between the latencies of the two systems; $T > 0$ indicates Optane has smaller latency, while $T < 0$ indicates Flash SSD has smaller latency. Specifically, $T = \frac{L_{higher} - L_{lower}}{L_{lower}}$. As shown, Optane SSD and Flash SSD outperform each other in different cases.

For read-only workloads, Optane SSD has lower latency than Flash SSD when request size and queue depth are small. Specifically, when the request size $\leqslant$ 16KB, Optane SSD is better than Flash SSD. Thanks to 3D XPoint memory's low access latency, the read latency from Optane SSD can be 8.4x faster than Flash SSD. However, when QD$>$ 8 and request size$>$ 16KB, Flash SSD achieves lower latency, by as much as 40%. This difference occurs because the latency of Optane SSD increases linearly with higher queue depths (e.g., the latency of a 4KB random read with QD$=$ 64 is 8x slower than QD$=$ 8 and is 11x slower than QD$=$ 1).



(a) Read



(b) Write

Figure 3.1: **Average Latency of Random Workloads, Optane vs. Flash.** *This figure compares average latency of random reads (a) and writes (b) between Optane SSD and Flash SSD. We show results for workloads with varying queue depths (x axis) and request sizes (y axis). The temperature* T *in each rectangle represents the scaled difference between Flash and Optane' latencies;* $T > 0$ *indicates Optane has lower latency (optane better), while* $T < 0$ *indicates Flash has lower latency (flash better). In particular,* $T = \frac{L_{higher} - L_{lower}}{L_{lower}}$.

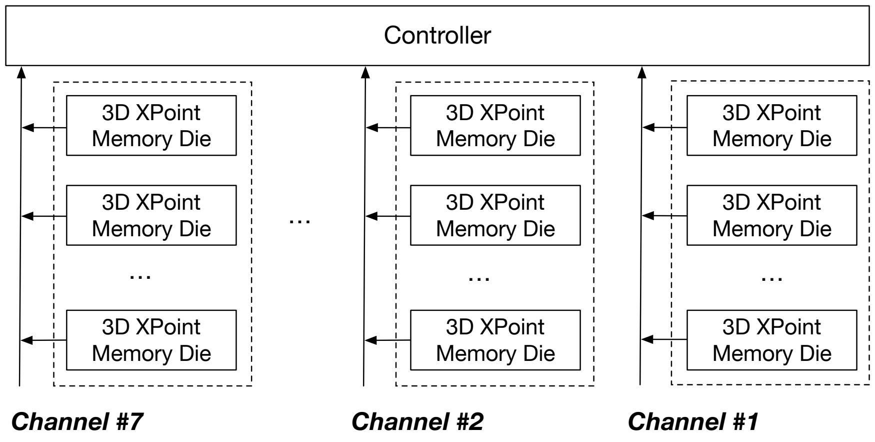
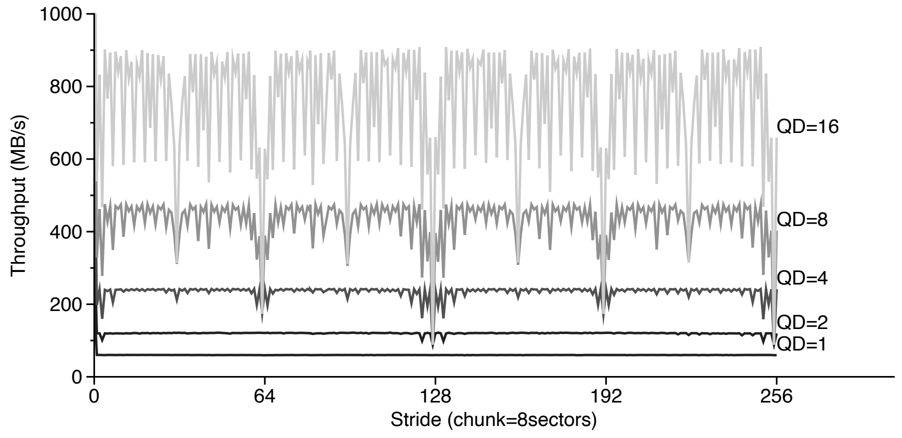


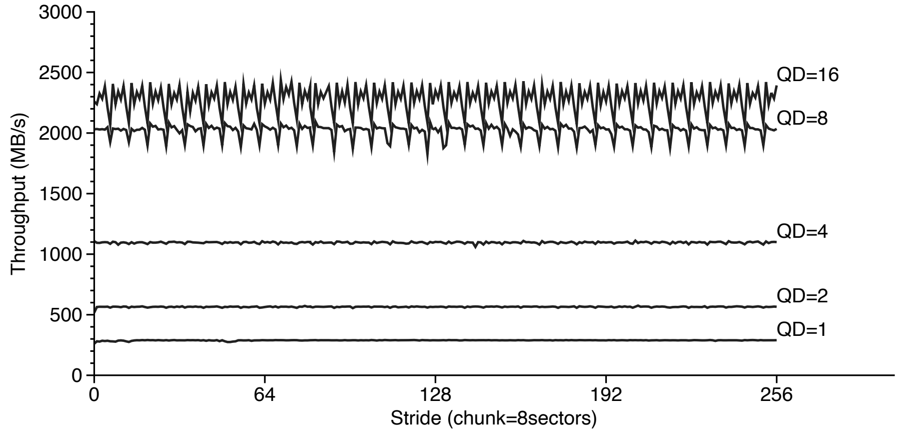
Figure 3.2: **Optane SSD RAID-like Architecture.** *This figure shows the Optane SSD's conceptual RAID-like internal architecture. Optane SSDs contain 3D XPoint Memory dies that are linked to a number of channels.*

Similarly, for write-only workloads, Optane SSD has lower latency for small requests and low QD, while Flash SSD is again better in the opposite cases; however, the difference for writes is not as high as for reads (Optane is up to 1.7x faster than Flash). This result is due to Flash SSDs log-structured layout and buffering optimizations. Flash SSD outperforms Optane SSD when request size$>$ 4KB and QD$>$ 2, which includes most of our tested workloads.

The device's internal parallelism dictates its behavior when serving workloads with high request scale. We are thus motivated to uncover the internals of the Optane SSD. Like Flash SSD, Optane SSD utilizes a RAID-like organization of memory dies (Figure 3.2). Through a fine-grained experiment [110, 122] we examine a critical parameter for internal parallelism: the interleaving degree, or number of channels. We maintain a read stream with stride S from the devices, where S is the distance between two consecutive chunks (a chunk is 4KB or 8 sectors as shown in Section 3.1.6). Figure 3.3 presents the throughput of workloads with various strides. An individual line represents workloads with the same QD.



(a) Flash SSD



(b) Optane 905P SSD

Figure 3.3: **Detecting Flash and Optane SSD Device Internal Interleaving Degree.** *This figure shows the interleaving degree detecting results for Flash (a) and Optane (b) SSDs. We keep a read stream from the devices with stride S; S is the distance between two consecutive chunk reads (4KB). We plot read stream throughputs (y axis) with various strides (x axis).*

Optane SSD has a significantly smaller interleaving degree (7) than Flash SSD (128). In Figure 3.3, the distance between the lowest dips in each line indirectly indicates the interleaving degree of the device. For Optane SSD, we observe the pattern is 7, though the difference between dips is not visible until QD= 8. Our finding agrees with the hardware description of Optane SSD [22]: it has a controller connected to seven channels, each of which is connected to memory dies.

The tested Flash SSD reaches its lowest throughput every 128 chunks. For high queue depths (e.g., 16), it presents a richer pattern with dips at different levels, indicating copious levels of parallelism (channel, package and die).

Overall, **Optane SSD has limited internal parallelism, compared to Flash SSD**. This characteristic explains the high latency of Optane SSD for workloads with a large request scale, and supports the need to access the Optane SSD with a low request scale.

The limited internal parallelism of the Optane SSD impacts its throughput in two ways. First, the tested Flash SSD achieves larger maximum read (3500MB/s) and write (2700MB/s) bandwidth than the Optane SSD (2500MB/s); it outperforms Optane SSD serving workloads with large request scale. Flash SSD's richer internal parallelism enables it to serve more parallel requests. Second, Optane SSD can achieve peak throughput when serving small requests with low queue depth. This is due to Optane SSD's limited internal parallelism which requires only a small number of requests to utilize all of its resources. Hence, the rule to access the device at a low scale not only guides users to obtain low access latency but also is enough to achieve full bandwidth.

The influence of contention in Optane SSD is modest compared to that in Flash SSD. The dips in Figure 3.3 are due to concurrent requests contending for shared resources (e.g., channels). In Flash SSD, contention significantly restricts overall throughput; for example, with

QD $= 16$, $S = 127$ chunks, read throughput is 88 MB/s, which is only 6% of the maximum throughput with the same queue depth. Although parallel requests to Optane SSD can also introduce contention (limiting throughput to within 86% of maximum), this influence is much less than that in Flash SSD.

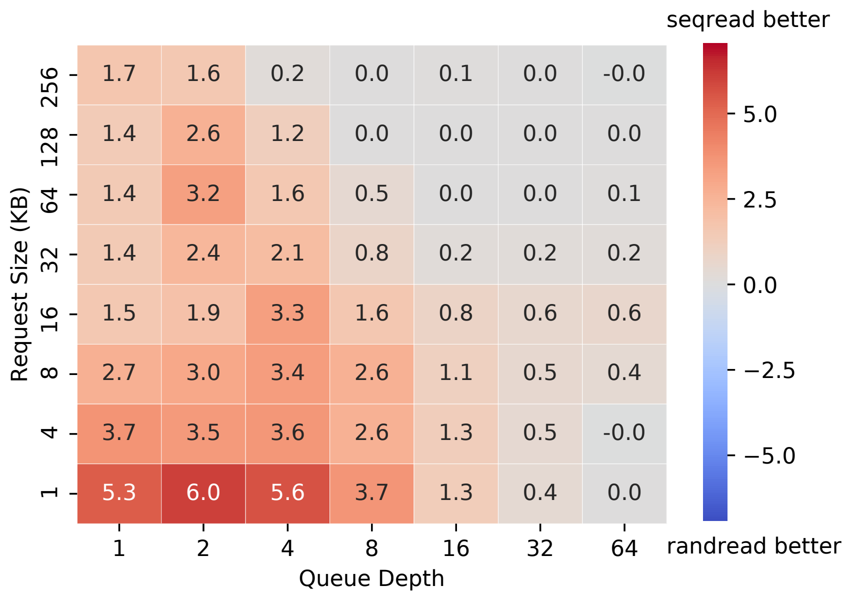
3.1.2 Random Access is OK

With hard drives or SSDs, clients often expect better performance from sequential than random accesses. With Optane SSD, this is no longer true. **Optane SSD is a random access block device**, where clients can observe the same performance for random and sequential workloads.

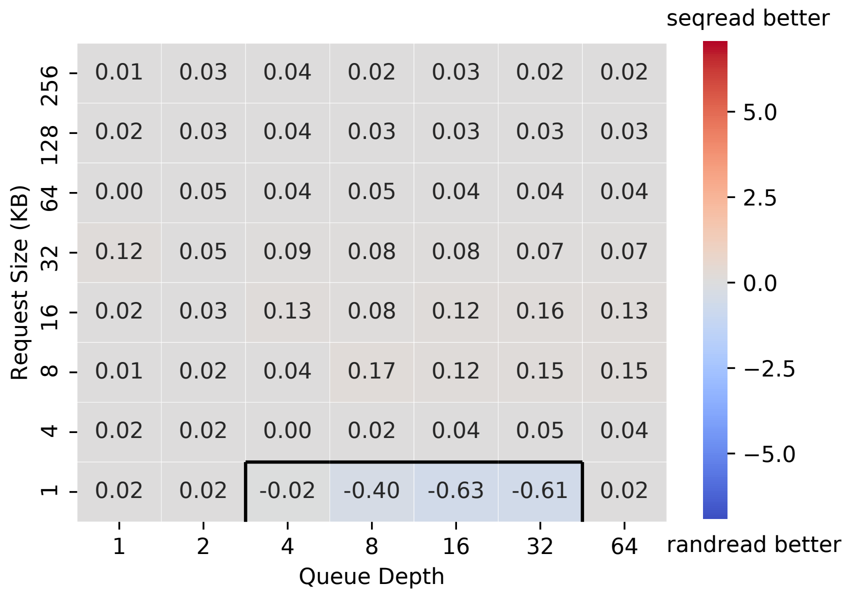
We study Optane SSD's average latency when serving random and sequential workloads; throughput and tail latency yield similar results. Each workload maintains four worker threads, while each thread issues IOs randomly or in a sequential stream. As shown in Figures 3.4 and 3.5, on Optane SSD, for requests $>$ 1KB, random and sequential workloads achieve comparable performance. The difference between sequential and random latency is within 17% for reads and within 5% for writes.

In contrast, Flash SSD prefers sequential over random reads, especially at a low request scale; sequential reads can be 7x faster. Flash SSD achieves much better sequential performance due to prefetching and simplified address translation. For writes, Flash SSD presents similar random and sequential latency due to log-structuring. However, as we will show in Section 3.1.7, log-structuring introduces significant overhead when the device fills; Optane does not have such concerns.

The Random Access is OK rule in Optane SSDs occurs due to the ability to perform in-place updates in 3D XPoint memory. In Optane SSD, there is no difference in address translation costs for random versus sequential workloads; in Section 3.1.7, we will show that the mapping policy in Optane SSD is based on logical addresses. In addition, as indicated

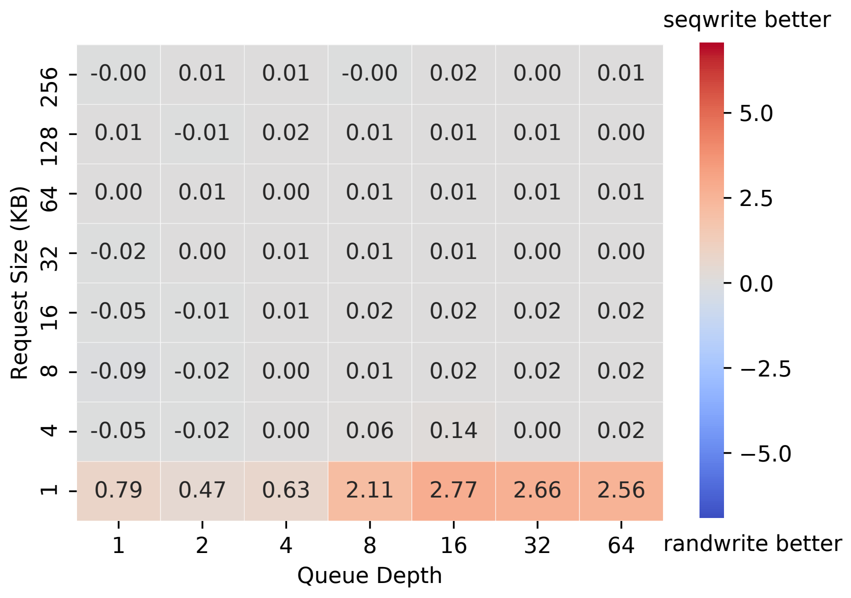


(a) Flash-based SSD

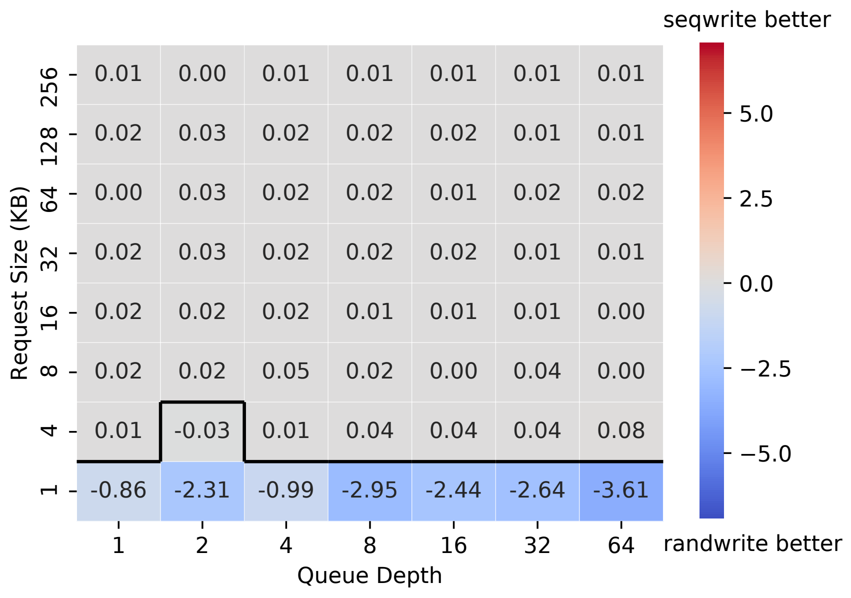


(b) Optane 905P SSD

Figure 3.4: **The Difference Between Random and Sequential Read Latency in Flash and Optane SSDs.** *This figure shows the performance difference between random and sequential reads in Flash (a) and Optane (b) SSDs. We use read workloads with varying request sizes (y axis) and queue depths (x axis). Similar to Figure 3.1, the temperature* T *in each rectangle represents the scaled difference between random and sequential reads' latencies;* $T > 0$ *indicates sequential read has lower latency (seqread better), while* $T < 0$ *indicates random reads has lower latency (randread better).* $T = \frac{L_{higher} - L_{lower}}{L_{lower}}$.



(a) Flash-based SSD



(b) Optane 905P SSD

Figure 3.5: **The Difference Between Random and Sequential Write Latency in Flash and Optane SSDs.** *This figure shows the performance difference between random and sequential writes in Flash (a) and Optane (b) SSDs. We use the same setup as Figure 3.4, but the workloads are writes instead of reads.*

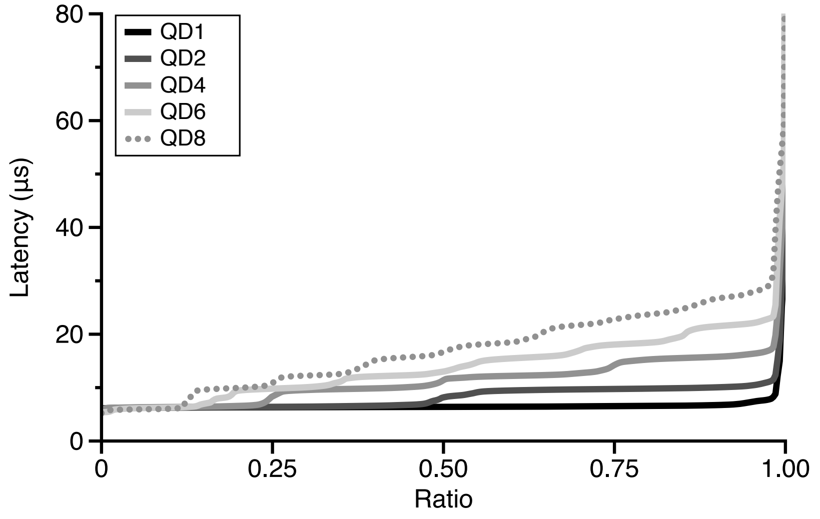


Figure 3.6: **Latency Distribution of Parallel Sector Reads to a Chunk (4KB) in the Optane SSD.** *In this experiment, we randomly select chunks (4KB) from the Optane SSD, then perform* P *parallel reads to different sectors (512B) within one chunk. We plot the latency distribution for different values of* P.

by our read latency study, there is no prefetching for sequential reads within Optane SSD.

Workloads with 1KB requests are special on Optane SSD compared both to other request sizes and to Flash SSD. We investigate this in the next rule.

3.1.3 Avoid Crowded Accesses

The Optane SSD contains shared resources (e.g., channels). To avoid contention, the Avoid Crowded Accesses rule dictates that clients of Optane SSD should never issue parallel accesses to a single chunk (4KB). We uncover this rule by investigating the 1KB workload performance shown in Figure 3.4 and 3.5. In Optane SSD, sequential 1KB accesses can increase latency by 63% for reads and by 3.6x for writes, compared to random 1KB accesses.

We study the difference between random and sequential accesses for small requests by performing parallel accesses to a single 4KB chunk. In this experiment, we randomly choose chunks from the device, then issue P parallel reads to different sectors within one chunk. Figure 3.6 presents the distribution of latencies for different values of P. We observe a "stair" pattern for QD> 1. For each line, the number of levels equals the queue depth and the steps occur at evenly-spaced intervals.

This pattern indicates queuing and/or contention across the issued parallel requests. **Parallel small requests to a single chunk introduce contention**. This experiment illustrates why sequential 1KB workloads have worse performance than random 1KB workloads: although random 1KB accesses *may* introduce contention, sequential 1KB accesses *must* introduce contention.

3.1.4 Control Overall Load

To achieve optimal latency from Optane SSD, the client must control the overall load of both reads and writes. This rule indicates distinct performance characteristics between Optane SSD and Flash SSD.

We discover this rule by looking into the performance of Optane SSD serving mixed reads and writes. In the experiment, we issue random 4KB requests, varying the percentage of writes from 0% to 100%, with QD$= 64$ (large enough to achieve full throughput for both Optane SSD and Flash SSD). Figure 3.7 shows the access latency of Optane SSD and Flash SSD. **Within Optane SSD, reads and writes are treated equally.** Specifically, on Optane SSD, each type of request is served with the same latency and the latency is related to the overall load, not to the percentage of writes.

Flash SSD exhibits distinct characteristics for read- versus write-dominated workloads. On the left side of Figure 3.7, for Flash SSD, the read latency is similar to that in read-only workloads with the same queue depth (38% slower than Optane SSD). However, the write latency is simi-

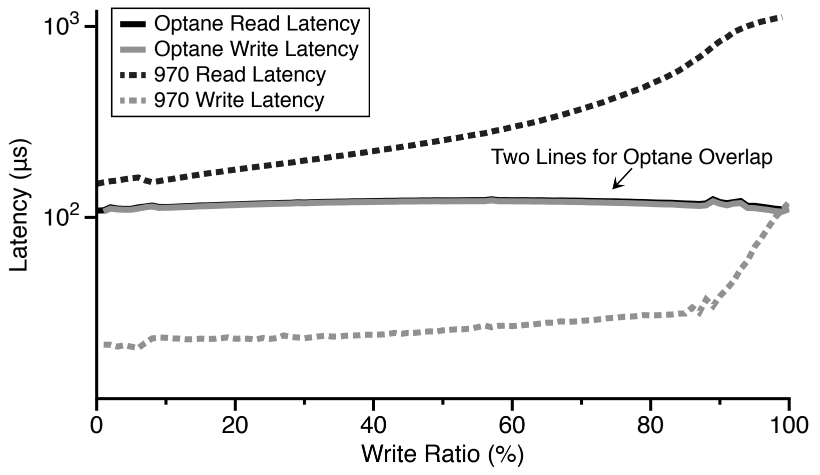


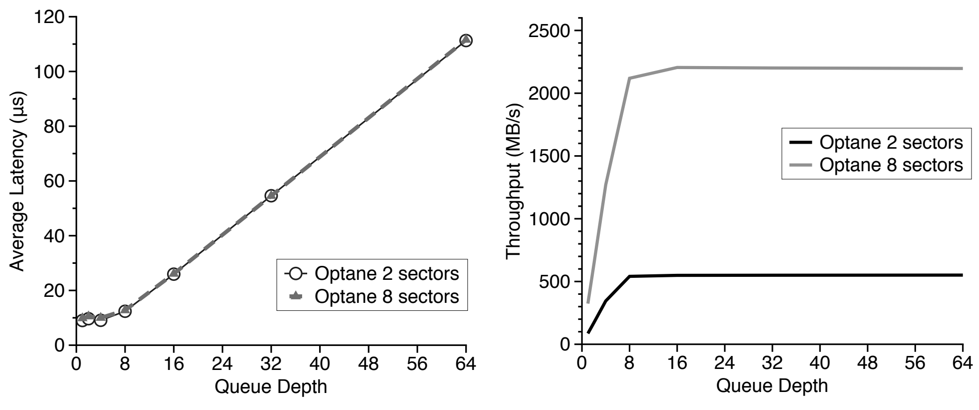
Figure 3.7: **Latency of Mixed Reads and Writes in the Flash and Optane SSD.** *This figure presents the request latency of mixed 4KB read and write workloads in Flash and Optane SSD. We show the average latency (x axis) of reads and writes as the ratio of writes changes (x axis).*

lar to that of a pure-write workload with very low queue depth (and only 19% of that on Optane SSD). With an increasing number of writes, reads to Flash SSD achieve poor latency due to the influence of writes; when the workload is write-dominated, read latency can be as high as 1.1ms (10x Optane access latency).

3.1.5 Avoid Tiny Accesses

Does the byte-addressability of 3D XPoint memory enable efficient tiny accesses to Optane SSD? We answer this question with our rule to Avoid Tiny Accesses: to exploit bandwidth of the SSD, the client must not issue requests less than 4KB.

Figure 3.8 shows the latency and throughput of random reads less than 4KB, with separate lines for two sectors and eight sectors requests. As shown, the latency of two sector requests is the same as eight sector (4KB) requests. However, the throughput of tiny requests is limited by



(a) Latency (b) Throughput

Figure 3.8: **Optane SSD Performance of Tiny Accesses.** *This figure shows the latency (a) and throughput (b) of random 1KB (2 sectors) reads. For comparison, we also show random 4KB (8 sectors) read performance.*

the maximum IOPS supported by Optane SSD (575K); for 1KB requests, throughput is only 20% of the full bandwidth of the device. Given these two results, **there is no benefit in issuing requests smaller than 4KB to Optane SSD**.

3.1.6 Issue 4KB Aligned Requests

To achieve the best latency, requests issued to Optane SSD should always align to eight sectors. We present the difference between aligned and misaligned requests. In the experiment, we measure the latency of individual read requests (QD= 1); each read is issued to a position A+*offset*, where A is a random position aligned to 32KB and *offset* is a 512-byte sector within that 32KB.

Figure 3.9 shows the read latency of requests issued to different offsets, averaged over a half million requests to the same offset. Each line represents a workload with a different request size. In contrast to what one might expect given 3D XPoint's byte-addressability, **Optane SSD fa-**

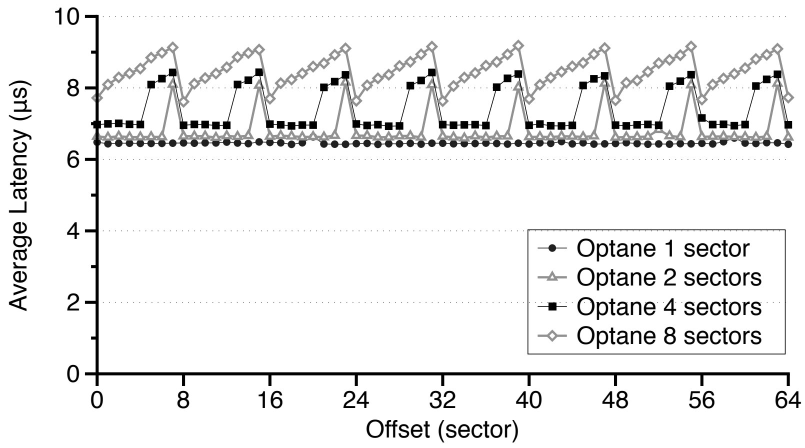


Figure 3.9: **Identifying Influence of Misalignment in Optane SSD.** *This figure shows the latency (y axis) of reads to varying offsets (x axis) to 32KB large chunks. Each read is sent to a position A+offset; A is a random position aligned to 32KB and offset is a 512B sector within that 32KB.*

vors aligned requests. Requests of one sector have the same latency no matter the offset, and larger requests aligned to eight sectors always get the lowest latency. For a request crossing the boundary of 4KB, its latency is linearly correlated to the part it issues to the second chunk after the boundary. The difference between the high and low latencies of 4KB requests is 21%. Note that the eight-sector chunk here is not related to a concept like a page or block in Flash SSD.

3.1.7 Forget Garbage Collection

We now study the long-term performance of Optane SSD. First, we explore its performance when the device gets full. According to our experiments, there is no need to worry about garbage collection in Optane SSD.

We examine sustained 4KB random and sequential writes on Optane SSD and Flash SSD over three hours. The device is completely unmapped using the trim command before each experiment; the two devices tested

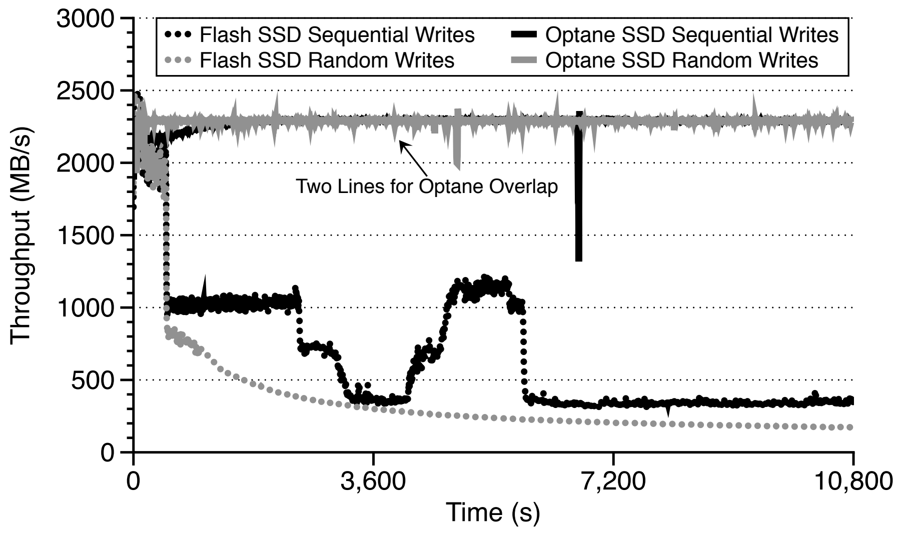
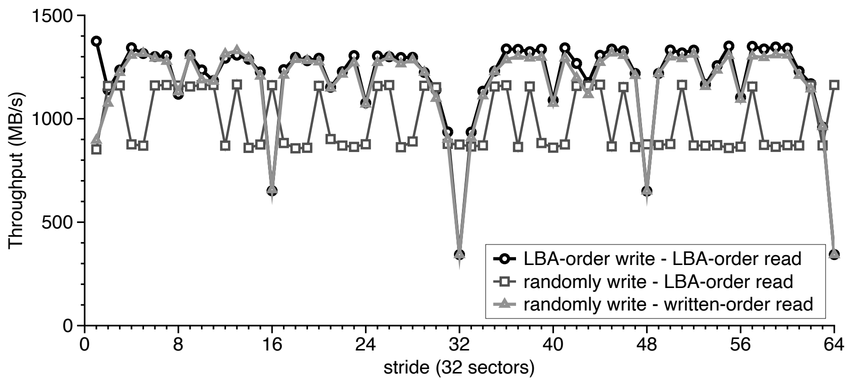


Figure 3.10: **Sustained Write Throughput, Flash vs. Optane.** *This figure shows the sustained 4KB random and sequential write throughput on Optane and Flash SSD.*

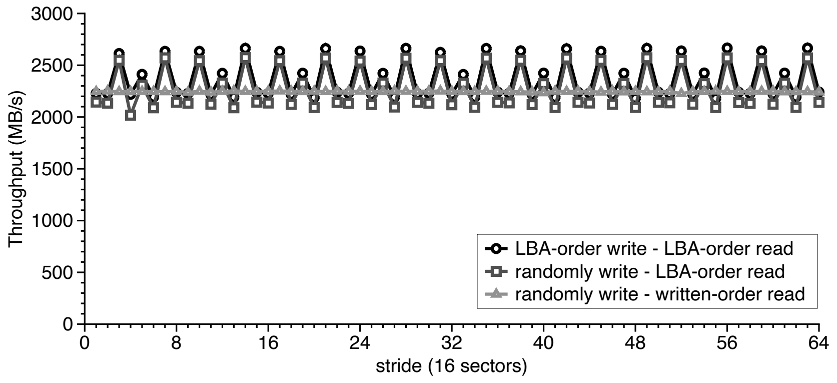
share a similar capacity (960GB vs. 1TB). We maintain QD= 32 for each workload.

Figure 3.10 presents sustained write performance. For Flash SSD, after the device becomes full, write throughput drops significantly because subsequent writes constantly trigger garbage collection. After about 6000 seconds, write throughput stabilizes: sequential throughput is around 350MB/s and random throughput is 170MB/s (only 7% of the maximum throughput). The throughput of sequential writes is better than random because of lower garbage collection cost. **Different from Flash SSD, Optane SSD maintains maximum throughput for sustained writes**. The flat throughput for Optane SSD indicates no cost for garbage collection.

Finally, we study the mapping policy (LBA$\rightarrow$PBA) in Optane SSD by comparing three workload variations. The first is the same workload as for the interleaving experiments in Figure 3.3: blocks are first written in logical address order and then read back in that same LBA order (LBA-order write:LBA-order read). The second workload preconditions



(a) Flash-based SSD



(b) Optane 905P SSD

Figure 3.11: **Detecting Flash and Optane SSD Internal Address Mapping Policy.** *This figure shows Flash (a) and Optane (b) SSD read throughput with three different precondition and read orders.* ***LBA-order write - LBA-order read****: the same as Figure 3.3: blocks are first written in logical address order and then read back in that same LBA order.* ***Randomly write - LBA-order read****: we precondition the working zone with random writes.* ***Randomly write - written-order read****: we precondition with random writes and then reads in the order in which the chunks were written.*

the working zone with random writes (random write:LBA-order read). The third workload preconditions with random writes, but then reads in the order in which the chunks were written (random write:written-order read). Figure 3.11 shows the throughput of the three workloads.

For Flash SSD, when the read order does not match the write order, the throughput pattern disappears; therefore, its mapping policy is not based on LBA. Flash SSD uses a mapping policy based on written-order (log-structured) and therefore the throughput pattern only occurs when we read according to the written order; this is why Flash SSD requires garbage collection. Optane SSD behaves quite differently; no matter how we precondition the device, the pattern occurs when reading according to LBA. Hence, **Optane SSD likely adopts LBA-based mapping.** This design is enabled by 3D XPoint memory's capability to perform in-place updates. As a result, Optane SSD doesn't require garbage collection and can deliver sustainable performance over time.

3.2 Discussion and Implications From the Contract

3.2.1 Flash vs. Optane SSD

According to our Optane SSD characterization, Flash and Optane SSD have some similarities but differ significantly in others.

Optane SSD, like Flash SSD, cannot support efficient small accesses (4KB). Developers should keep this in mind when distinguishing Optane SSD from PM DIMMs. Internal data chunk alignments are also present in both Optane SSD and Flash SSD (4KB). Finally, both the Optane SSD and the Flash SSD employ a RAID-like architecture of internal channels linked to the device controller.

However, Optane SSD and Flash SSD differ in four important ways:

I **Internal Parallelisms**: When compared to Flash, Optane SSD has much lower access latency but much less internal parallelism. Optane SSD 4KB accesses can be orders of magnitude faster than Flash SSD. However, as we revealed, current Optane SSD devices opt for much less parallelism. We believe that this design decision has complicated implications for Optane SSD users.

II **Random vs. Sequential:** Optane SSD is a true random access block device, with identical random and sequential read and write performance. This is a significant distinction between Optane and Flash.

III **Read-Write Interferences:** Another significant difference revealed by our experiments is the difference in read-write interference patterns between Optane and Flash. We discovered that Optane treats read and write equally, whereas Flash SSD frequently prioritizes writes over reads, potentially resulting in significant write to read interferences in Flash.

IV **LBA to PBA mapping:** Finally, our sustained workloads and experiments revealing Optane's internal LBA to PBA mapping experiments show that: Optane SSD no longer performs log-structured writes, and garbage collection is no longer an issue.

Overall, Optane SSD differs significantly from Flash SSD; Optane SSD is more than just faster Flash devices. Following that, we will discuss the exciting opportunities that these devices bring, as well as the implications for evolving systems for this type of new device.

3.2.2 Implications From the Contract

The following implications can be drawn from our unwritten contract. We have two audiences in mind; first, those who design systems for Optane SSD; second, those who combine Flash and Optane in a hybrid setting.

The Random Access is OK rule suggests possible restructuring of external data structures on Optane SSD. Previous designs try hard to convert unstructured accesses into sequential ones, which is now less necessary on Optane SSDs. Applications which behave poorly on Flash thus become potential consumers of Optane. The No Crowded Accesses rule, No Tiny Access rule, and Alignment rule suggest pitfalls that fine-grained data structures must be aware on Optane SSDs.

Storage hierarchy management also needs careful design. This implication mainly comes from from the Low Request Scale rule: Flash SSD outperforms Optane SSD in some cases. Optane SSD provides low access latency for workloads with low request scale, while workloads with high request scale might prefer Flash SSD. In other words, the storage hierarchy, that many assumed when comparing Optane and Flash SSD, is not a hierarchy for all time. Management of such hierarchies necessitate careful rethinking. For workloads with mixed reads and writes, the Control Overall Load rule suggests that read-dominated workloads should be deployed on Flash SSD to achieve low write latency. However, write-intensive workloads prefer the Optane SSD, thus avoiding excessive read latency (influenced by writes in classic Flash SSD). Finally, our results for sustainable performance suggest that when devices become full (and thus would cause garbage collection on an SSD), Optane may be a better choice.

Finally, the unwritten contract introduces some open questions requiring discussion and future research:

I **Applicability of This Contract:** This contract targets the Intel Optane SSD 905P. It is not clear of the applicability of this contract; will this contract stay accurate for all PM-based SSDs? On the one hand, this contract needs to be verified once new devices are available. Better specifications (instead of simple best-case latency or throughput numbers) of the unwritten contracts of new types of devices, on the

Device Internal Aspects	Findings
Internal Parallelism	Seven channels
Potential Concurrent Access Contention	Minimal (vs. Flash SSD)
Internal Prefetching (for sequential access)	No
Read-Write Scheduling	Treating read/write equally
Efficient Access Granularity	4KB
Internal Layout Alignment	4KB
LBA -> PBA mapping	LBA-based mapping (no garbage collection)

Table 3.2: **Summary of Findings of Intel Optane SSD Internals.** *This table summarizes the device internal aspects that we investigated via micro-experiments, and our findings. LBA: logical block address. PBA: physical block address.*

other hand, should be provided to ease the development of new system stacks for these new devices.

II **Move to Optane SSD:** According to the contract, Optane SSD does not promise benefits for all applications. Which applications suit the move to Optane SSD naturally? What may need modification to exploit the benefits of Optane SSD? An analysis of applications on Optane SSD is required.

III **Rethink External Data Structure Design:** Do previous external data structures/algorithms get the optimal performance from Optane SSD? To answer this question, we think it is worth to first review "Is there any tradeoff we made to transform unstructured accesses into sequential ones?" This tradeoff likely exists in single machine graph processing systems. There is already recent work[132] following this path (but for NVMe SSD). On the contrary, can we move in-memory data structures to Optane SSD directly? A study is required. Previous work on PM [99, 178, 251] can be inspiring.

IV **Split Accesses:** Led by the unwritten contract, we think the guide to take advantage of both Flash SSD and Optane SSD is to split accesses to the device which best suit them. Some directions are interesting to look into. 1) At what granularity should we implement this splitting? Split workloads to devices? Or make the external data structure span devices? 2) How to support the splitting automatically? There can be challenges like dynamic workloads and access dependencies. Can machine learning play a role here? Related work includes [100, 177]

V **Wearing out:** There is limited open knowledge about the wearing out of 3D XPoint memory. How severe is it? How is wear-leveling performed in Optane SSD? How is it done in Optane DC PM? Those questions also require detailed future analysis of PM devices.

3.3 Conclusions

In this chapter, we analyzed a popular PM-based block device: the Intel Optane SSD. We formalize the rules (as summarized in Table 3.1) that Optane SSD users need to follow. We also provide experiments to present the impact when violating each rule, and examine the internals of Optane SSD to provide insights for each rule. As summarized in Table 3.2, we revealed key aspects of Intel Optane SSD such as internal parallelism, efficient access granularity, LBA->PBA address mapping policy etc. The unwritten contract provides implications and points to directions for potential research on PM-based devices. We believe that this study will serve as a foundation for evolving the system stack for Optane SSD devices.

4 Evolving Caching for PM Hierarchies

The modern storage hierarchies have been reshaped by PM-based devices. Following an understanding of PM device performance characteristics, in this chapter, we analyze classic caching on modern hierarchies with PM devices. We introduce Non-Hierarchical Caching (NHC), an evolved caching approach to manage modern hierarchies. This chapter is based on the paper, *The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus*, published in FAST 2021 [230].

In the first part of this chapter, we show that the decades-old caching principle of maximizing hit rates is insufficient in modern storage hierarchies. We quantitatively compare various device pairs in modern hierarchies, such as DRAM/PM/Low-latency SSD/Flash SSDs. Our analysis reveals a significant change in modern hierarchies: the performance difference between neighboring layers is now much smaller than in traditional hierarchies. We then investigate caching on such hierarchies using both performance modeling and measurements. We find that classic caching, which aims to direct as many accesses to the cache device as possible (commonly referred as the principle of maximizing the hit rate), is unable to fully utilize the significant available performance in capacity devices (e.g., PM) today. To manage PM hierarchies, a different approach is required.

Thus, in the second part of this chapter, we propose Non-hierarchical Caching (NHC), an enhanced caching approach. NHC augments classic caching by adding two new mechanisms: read around and admission rejection. Its central idea is to enable access offloading from cache to capacity devices when the cache device is saturated, allowing NHC to utilize both cache and capacity device performance. We also introduce a feedback-based offloading policy in NHC to determine the amount of offloading based on the workloads and devices in the hierarchies. We implement NHC in two systems: Orthus-CAS, a generic block-layer caching kernel module [108], and Orthus-KV, a user-level caching layer for an LSM-tree key-value store [168]. Through a variety of experimental studies, we demonstrate that NHC significantly improves caching performance in modern hierarchies with heavy workloads. We also show that NHC is robust to dynamic workloads.

In the following, we first present necessary background and motivation related to modern PM hierarchies in Section 4.1. We then characterize caching in traditional and modern hierarchies and describe the issue of classic caching in modern hierarchies (Section 4.2). Next, we describe the design of Non-hierarchical Caching (Section 4.3). Then, we describe NHC's implementation (Section 4.4) and present our evaluation (Section 4.5). Finally, we summarize and conclude (Section 4.6).

4.1 Background and Motivation

In this section, we discuss classic solutions to storage hierarchy management. We then review current and near-future PM devices and discuss how they alter the storage hierarchy.

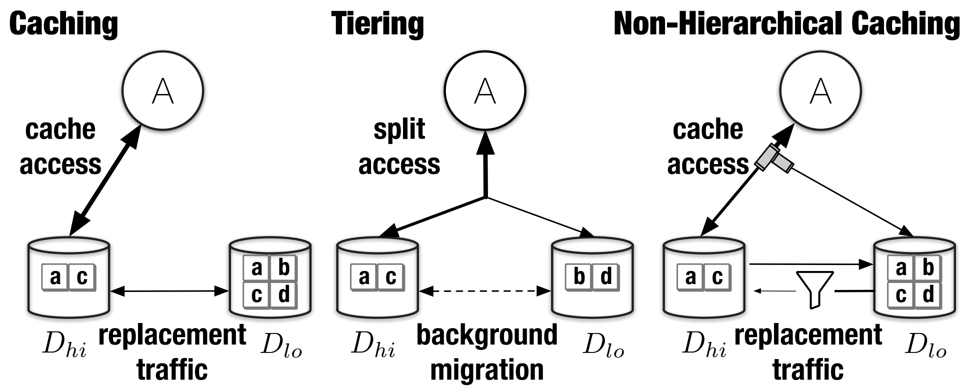


Figure 4.1: **Caching, Tiering, and Non-Hierarchical Caching.** *The figure shows the different approaches to managing a storage hierarchy. Caching copies data items to the performance layer upon a miss. Tiering splits access to each layer and migrates items in the background (on longer time scales). Non-hierarchical caching (§4.3), our new approach, offloads excess load at the performance layer to the capacity layer.*

4.1.1 Managing the Storage Hierarchy

A storage hierarchy is composed of multiple heterogeneous storage devices and policies for transferring data between those devices. For simplicity, we assume a two-device hierarchy, consisting of a *performance* device, D_{hi}, and a *capacity* device, D_{lo}; commonly, D_{hi} is more expensive, smaller, and faster, whereas D_{lo} is cheaper, larger, and slower.

Traditionally, two approaches have been used for managing such a hierarchy: *caching* and *tiering* (Figure 4.1). With caching, popular (hot) data is copied from D_{lo} into D_{hi} (e.g., on each miss); to make room for these hot data items, the cache evicts less popular (cold) data, as determined by algorithms such as ARC, LRU, or LFU [15, 171, 181, 192, 222, 250]. The granularity of data movement is usually small, e.g., 4-KB blocks.

Tiering [138, 161, 206], similar to caching, usually maintains hot data in the performance device. However, unlike caching, when data on D_{lo} is accessed, it is not necessarily promoted to D_{hi}; data can be directly served

Example	**Latency**	**Read** (GB/s)	**Write** (GB/s)	**Cost** ($/GB)
DRAM	80ns	15	15	~7
PM DIMM	300ns	6.8	2.3	~5
Low-latency SSD	10us	2.5	2.3	1
NVMe Flash SSD	80us	~3.0	~2.0	0.3
SATA Flash SSD	180us	0.5	0.5	0.15

Table 4.1: **Diversified Storage Devices.** *This figure compares diversified storage devices today. Data taken from SK Hynix DRAM(DDR4, 16GB), Intel Optane DC PM [37, 38], low-latency SSDs (Optane SSD 905P [40], Micron X100 SSD [56]), NVMe Flash SSD (Samsung 970 Pro [69, 70]) and SATA Flash SSD (Intel 520 SSD [41]). Low-latency and NVMe Flash SSD assume PCIe 3.0.*

from D_{lo}. Data is only periodically migrated between devices on longer time scales (over hours or days) and longer-term optimizations determine data placement. Tiering typically does such migration at a coarser granularity (an entire volume or a large extent[138]). While caching can quickly react to workload changes, tiering cannot do so given its periodic, coarser-granularity migration.

Both classic caching and tiering, to maximize performance, strive to ensure that most accesses are served from the performance device. Most caching and tiering policies are thus designed to maximize hits to the fast device. In traditional hierarchies where the performance of D_{hi} is significantly higher than D_{lo}, such approaches deliver high performance. However, with the storage landscape rapidly changing, modern devices have overlapping performance characteristics and thus it is essential to rethink how such devices must be managed.

4.1.2 Hardware Storage Trends

As shown in Table 4.1, storage systems are undergoing a rapid transformation with a proliferation of high-performance technologies, including persistent memory (e.g., 3D XPoint memory [1, 141]), low-latency SSDs (e.g., Intel Optane SSD [40], Samsung Z-SSD [71], and Micron X100

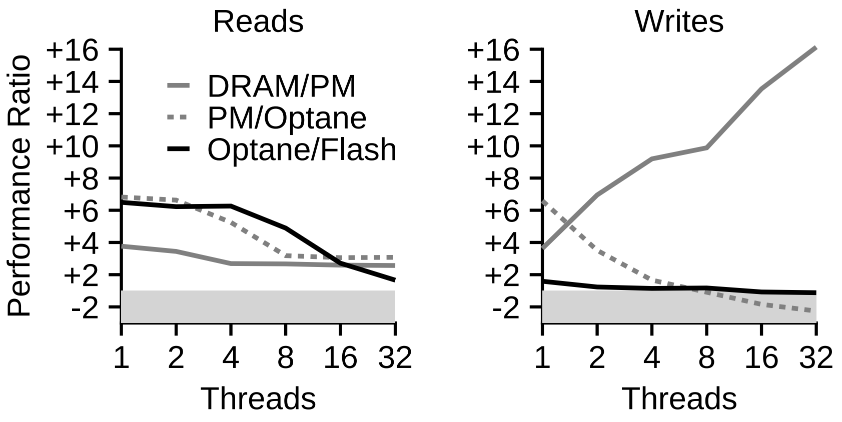


Figure 4.2: **Performance Ratios Across Modern Devices.** *This figure shows the ratio of throughput (y axis), for varying concurrency (x axis), across device pairings (DRAM/PM, PM/Optane, and Optane/Flash). We disable the cache prefetcher and use non-temporal stores for DRAM and PM. PM is used as App-Direct mode. Note there is no value between -1 and +1.*

SSD [56]), NVMe Flash SSDs ([69, 70]), and SATA Flash SSDs ([41]). Although a seeming ordering exists in terms of latency, bandwidth differences are less clear, and a total ordering is hard to establish.

To better understand the performance overlap of these devices, Figure 4.2 shows the throughput of a variety of real devices for both 4KB read/load and write/store while varying the level of concurrency. The figure plots the performance ratio between pairs of devices: DRAM/PM plots the bandwidth of memory (SK Hynix 16GB DDR4) vs. a single Intel Optane DC PM (128GB); NVM/Optane uses the DC PM vs. the Intel 905P Optane SSD; finally, Optane/Flash uses the same Optane SSD and the Samsung 970 Pro Flash SSD. For any pair X/Y, a positive ratio ($\frac{X}{Y}$) is plotted if the performance of X is greater than Y; otherwise, a negative ratio ($\frac{-Y}{X}$) is plotted (in the gray region).

For reads with low concurrency, one can see significant differences between device pairs. Thus, one might conclude that a total ordering exists. However, for reads under high concurrency, the ratios change dramatically. In the most extreme case, the Optane SSD and Flash SSD have nearly identical performance. For writes, the results are even more intriguing; because of the low performance of PM concurrent writes, in one case (PM/Optane), the ratio changes from much better under low load to much worse under high load.

To summarize, the following are the key trends in the storage hierarchy. Unlike the traditional hierarchy (e.g., DRAM vs. HDD), the new storage hierarchy may not be a hierarchy; the performance of two neighboring layers (e.g., NVM vs. Optane SSD) can be similar. Second, the performance of new devices vary depending upon many factors including different workloads (reads vs. writes) and level of concurrency. Managing these devices with traditional caching and tiering is no longer effective. Given our focus on improving caching approaches in this chapter, we next demonstrate the limitations of caching in modern hierarchies.

4.2 Characterizing Caching in Traditional and Modern Storage Hierarchies

We now explore caching in different storage hierarchies. Our goal is simple: to understand how caching performs in both traditional and modern hierarchies. In doing so, we hope to build towards a technique that addresses the limitation of caching when running on modern, complex devices and underneath a range of dynamic workloads.

For a deeper intuition, we first model caching performance. We then conduct an empirical analysis on real devices, filling in important details not captured by the model. We also model an approach that we call *splitting* to highlight the drawbacks of classic caching. In splitting, data is simply split across devices, and no migration is performed at run time. Splitting outperforms caching when accesses are optimally split between the performance and capacity devices. In contrast to caching and tiering, splitting is impractical: it is not suitable for workloads where popular items change over time; we use it only as a baseline to build up to our solution.

4.2.1 Modeling Caching Performance

We assume there are two devices, D_{hi} and D_{lo}, where each performs at a fixed rate, R_{hi} and R_{lo} ops/s; of course, real devices are more complex, with internal concurrency and performance that depends on the workload, but this simplification is sufficient for our purposes. Meanwhile, we assume that simultaneous access to these two devices does not cause contention (e.g., due to sharing interconnects).

We also assume that the workload has either little concurrency (i.e., one request at a time) or copious concurrency (i.e., many requests at a time). This allows us to bound the caching performance between these extremes. We assume that the workload is read only; this simplifies our analysis in that we do not account for dirty writebacks upon a cache replacement.

4.2.1.1 Model

We develop a model of caching performance based on hit rate, $H \in [0, 1]$. As stated above, we model two extremes: low and high concurrency. For one request at a time, the average time per request is:

$$T_{cache,1} = H \cdot T_{hit} + (1 - H) \cdot T_{miss} \tag{4.1}$$

T_{hit} is simply the inverse of the rate of the fast device, i.e., $T_{hit} = \frac{1}{R_{hi}}$; T_{miss} is the cost of fetching the data from the slow device and also installing it in the faster device, i.e., $T_{miss} = \frac{1}{R_{hi}} + \frac{1}{R_{lo}}$, or $\frac{R_{hi}+R_{lo}}{R_{hi} \cdot R_{lo}}$.

The resulting bandwidth is the inverse of $T_{cache,1}$:

$$B_{cache,1} = \frac{R_{hi} \cdot R_{lo}}{H \cdot R_{lo} + (1 - H) \cdot (R_{hi} + R_{lo})} \tag{4.2}$$

We now model concurrent workloads. Assume N requests. $H \cdot N$ are hits, $(1 - H) \cdot N$ are misses. Note that only misses are serviced by the slow device, whereas *all* requests must be serviced by the fast one (data admissions). The time to process N requests on the slow or fast device is:

$$T_{slow}(N) = N \cdot (1 - H) \cdot \frac{1}{R_{lo}} \tag{4.3}$$

$$T_{fast}(N) = N \cdot (1 - H) \cdot \frac{1}{R_{hi}} + N \cdot H \cdot \frac{1}{R_{hi}} = N \cdot \frac{1}{R_{hi}} \tag{4.4}$$

Total time is the maximum of these two, i.e., whichever device finishes last determines the workload time.

$$T_{cache,many}(N) = max(T_{slow}(N), T_{fast}(N)) \tag{4.5}$$

$$= max(N \cdot \tfrac{1-H}{R_{lo}}, N \cdot \tfrac{1}{R_{hi}}) \tag{4.6}$$

Dividing by N (not shown) yields the average time per request. Finally, the bandwidth can be computed, as it is the inverse of the average time per request:

$$B_{cache,many} = \frac{1}{max(\frac{1-H}{R_{lo}}, \frac{1}{R_{hi}})} \tag{4.7}$$

We model splitting performance based on the split rate, $S \in [0, 1]$, which determines the fraction S of requests serviced at D_{hi}; the remaining requests $(1 - S)$ are served at the tier D_{lo}. Compared to caching, splitting eliminates the cost of installing misses on the faster device. Its throughput can be computed as follows (in a similar way as caching, note the different formula for D_{hi}):

$$B_{split,1} = \frac{1}{\frac{1-S}{R_{lo}} + \frac{S}{R_{hi}}} \tag{4.8}$$

$$B_{split,many} = \frac{1}{\max(\frac{1-S}{R_{lo}}, \frac{S}{R_{hi}})} \tag{4.9}$$

4.2.1.2 Model Exploration

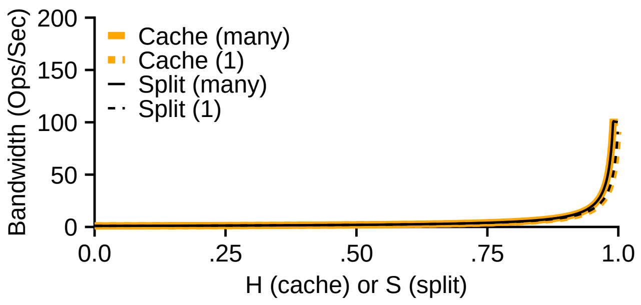
We explore different parameter settings with our model. Figure 4.3 shows the results for four settings, starting with a large difference in performance between D_{hi} and D_{lo}, and then slowly increasing the performance of D_{lo}.

The first graph shows a traditional hierarchy where the performance of D_{hi} is much (100×) higher than the performance of D_{lo}. This graph shows that both caching and splitting can deliver high performance on traditional hierarchies. The key is to direct as many requests as possible to D_{hi}. Caching and splitting perform well if nearly all requests hit in D_{hi}. Even with 80% hit/split rate, overall performance is quite low, as the slow device dominates.

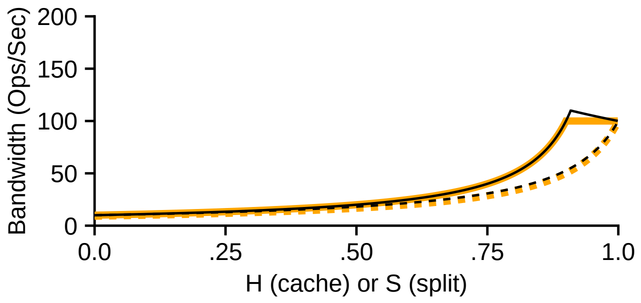
The next graph (upper right) examines a case where the performance ratio between the devices is still high (10×). Optimizing for a high hit/split rate still works well. Note the slight difference between the low and high concurrency cases; with higher concurrency, these approaches can achieve peak performance even with slightly less than a perfect hit rate, as outstanding requests hide the cost of misses.

The next two graphs represent modern hierarchies where the performance of D_{hi} is closer to that of D_{lo} (D_{hi} delivers bandwidth either 2× D_{lo} or equal to it). We make two important observations from these graphs.

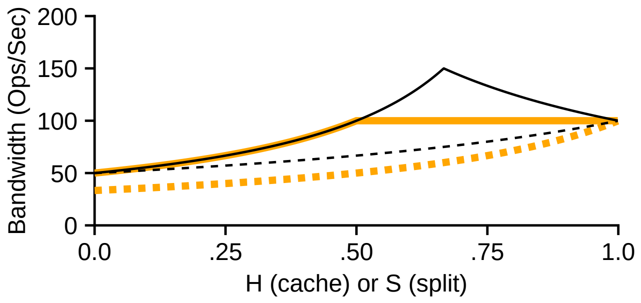
First, classic caching is limited by the performance of D_{hi} and cannot realize the combined performance of both devices. Even with a 100%



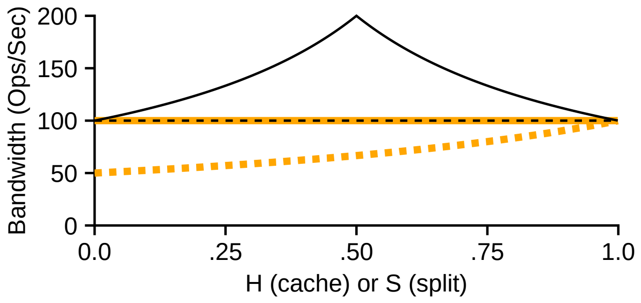
(a) Device Ratio 100 : 1



(b) Device Ratio 100 : 10



(c) Device Ratio 100 : 50



(d) Device Ratio 100 : 100

Figure 4.3: **Exploring Modeled Cache Performance.** *This figure shows model-predicted throughput (y axis) for caching and splitting at various hit/split ratios (H or S, x axis). We investigate a range of different device performance levels (subfigure a,b,c,d). We show performance for workloads with high ("many") and low ("1") concurrency. The faster device performs at a fixed rate of 100 ops/sec.*

hit ratio, caching can only deliver 100 ops/sec as it does not utilize the bandwidth of D_{lo}. Splitting (with an optimal split rate) significantly outperforms caching, exposing critical limitations of caching in modern hierarchies.

Second, in modern hierarchies, maximizing the number of requests served by D_{hi} does not always yield the best performance. Consider the case where D_{hi} is $2\times$ faster than D_{lo}. With copious concurrency, when about two-thirds of the requests are directed to D_{hi}, splitting realizes the aggregate bandwidth of D_{hi} and D_{lo}. Increasing the split rate further only degrades performance. Thus, in modern hierarchies, instead of maximizing the hit or split rate, the key is to find the right proportion of requests that must be sent to each device.

4.2.2 Evaluation with Optane DC PM and Optane SSD

Next, we demonstrate that the observations from our model hold for real storage stacks. We use one traditional hierarchy consisting of DRAM and a Flash SSD [69]. We also use two modern stacks: first, PM (Optane DC PM 128GB) and an Optane 905P SSD; second, an Optane SSD and a Flash SSD. We use these hierarchies to cover a wide range of performance differences; meantime, Optane DC PM and Optane SSD are the most popular emerging devices nowadays. While there could be many hierarchies (e.g., with different versions of these devices), we believe our hierarchies are adequate to validate our modeling and draw meaningful implications for our designs.

For these experiments, we have implemented a new benchmarking tool, called the *Hierarchical Flexible I/O Benchmark Suite* (*HFIO*). HFIO contains a configurable hierarchy controller that implements caching and splitting. HFIO uses the LRU-replacement policy for caching. HFIO generates synthetic workloads with a variety of parameters (e.g., mix of reads and writes, locality, and the number of concurrent accesses). HFIO pre-

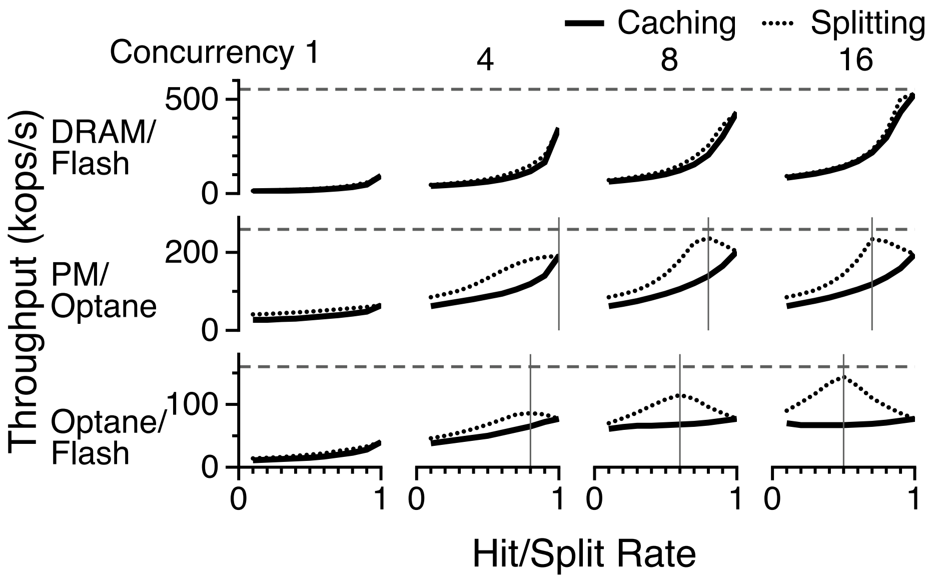


Figure 4.4: **Measured Performance of Caching and Splitting.** *This figure shows the caching and splitting throughput of read-only workloads. We plot results for three hierarchies (DRAM/Flash, PM/Optane, Optane/Flash) and four workloads with different concurrency levels (1, 4, 8, 16). Horizontal dotted lines represent the combined bandwidth of both devices (the maximum possible throughput).*

cisely controls the caching layer size and access locality to obtain a desired hit rate. We fix the block size to 32 KB and consider only random accesses. We run our experiments on an Intel Xeon Gold 5218 CPU at 2.3GHz (16 cores), running Ubuntu 18.04. All experiments ran long enough to fill the cache and deliver steady-state performance.

We begin by replicating the results from our model by running read-only workloads and measuring the throughput. Figure 4.4 shows the results on three hierarchies and workloads with different levels of concurrency. First, in the traditional hierarchy (DRAM+Flash SSD, the first row of Figure 4.4), as expected, both caching and splitting can achieve high performance. Caching and splitting perform similarly, exactly as our model predicted (Figure 4.3, 100:1 and 100:10 cases).

The second two rows of Figure 4.4 show that caching in new storage hierarchies (e.g., PM+Optane, Optane+Flash) behaves much differently than in the traditional hierarchy. With low concurrency (1 or 4), the caching device (i.e., Optane DC PM or Optane SSD) is not fully utilized and thus optimizing the hit/split rate still improves performance. However, for workloads with more concurrency, maximizing the hit/split rates does not lead to peak performance in either of the PM+Optane or Optane+Flash hierarchies. In these situations, capacity devices such as Optane SSD provide substantial performance compared to their caching layers (e.g., DC PM). Splitting (with an optimal split rate) can thus deliver significantly greater performance than caching.

Our experiments with real devices reveal several complexities that the models do not: the optimal split rate depends upon several factors. From Figure 4.4, we can see that the optimal split rate varies significantly from one device to another and with the level of parallelism of the workload. Write ratios also influence the optimal split rate. As shown in Figure 4.5, for Optane+Flash, the optimal split rate for a read-heavy workload is 90%, while it is about 60% when the workload is write-heavy. This change occurs because the difference between the write performances of Optane and Flash is smaller than the difference between their read performances. We observe similar results for the PM+Optane hierarchy.

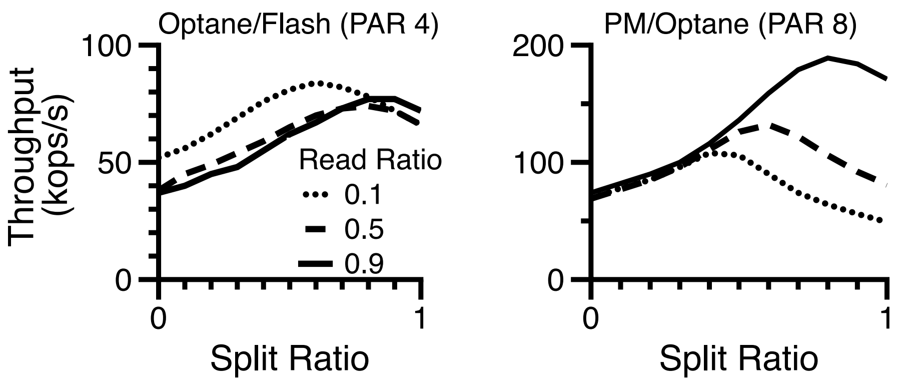
Summary and Implications: Our performance characterization (modeling and evaluation) of caching provides important lessons for our design. Classic caching is no longer effective in modern hierarchies: it does not exploit the considerable performance that can be delivered by the capacity layer. With high hit rates and when the cache layer is under heavy load, some of the requests can be offloaded to the capacity device. Such high hit rates and heavy load are quite common in production caching systems. For instance, a recent study at Twitter showed that eight out of the ten Twemcache clusters have a miss ratio lower than 5% [239]. Studies have also shown that cache layers often experience heavy load (i.e., they are bandwidth saturated) [78, 160].

In the modern hierarchy, the capacity layer can offer substantial performance and should thus be exploited in such situations. An alternative solution is to increase the number of cache devices in the hierarchy; however, this approach can be prohibitively expensive as performance devices are more costly. In contrast, offloading requests to the capacity layer offers a more economic way to realize significant improvements. Such an offloading approach can deliver the aggregate performance of all devices by optimally splitting the requests to each device. For the offloading approach to work well, it is essential to dynamically adjust the split rate because the optimal rate varies widely in modern hierarchies depending upon factors such as write ratios and level of concurrency.

We note that classic tiering (which also aims to direct most requests to the performance layer) suffers from similar shortcomings as caching in modern hierarchies. In this work, we focus on improving caching for two main reasons. First, getting tiering to optimally split accesses is fundamentally hard. Migration or replication to match the current optimal split in tiering may hurt performance. In contrast, caching can readily bypass cache hits to capacity devices; copies of hot data are always available on both devices. Second, we believe there are many scenarios where caching may be the only suitable solution and tiering may not be appropriate. For instance, applications can only use DRAM as a cache when persistence is required and cannot tier in the DRAM+PM hierarchy. We believe many systems use caching for such reasons and an approach that improves upon classic caching can be beneficial for many such systems.

4.3 Non-Hierarchical Caching

We present *non-hierarchical caching* (NHC), a caching framework that utilizes the performance of devices that would have been treated as only a



(a) Optane SSD + Flash (b) PM + Optane SSD

Figure 4.5: **Splitting Performance with Mixed Reads and Writes Workloads.** *The figure shows the performance of splitting with mixed read and write workloads; PAR: workload parallelism/concurrency. We show results for Optane SSD + Flash hierarchy (a) and PM + Optane SSD hierarchy (b).*

capacity layer with classic caching. NHC has the following **goals**:

(i) Perform as well or better than classic caching. Classic caching optimizes the performance of a storage hierarchy by optimizing the performance from the higher-level device, D_{hi}; this performance is optimized by finding the working set that maximizes the hit rate. NHC should degenerate in the worst-case to classic caching and should be able to leverage any classic caching policy (e.g., eviction and write-allocation).

(ii) Require no special knowledge or configuration. NHC should not make more assumptions than classic caching. NHC should not require prior knowledge of the workload or detailed performance characteristics of the devices. NHC should be able to manage any storage hierarchy.

(iii) Be robust to dynamic workloads: Workloads change over time, in their amount of load and working set. NHC should adjust to dynamic changes.

The main idea of NHC (Figure 4.1) is to offload excess load to capacity devices when doing so improves the overall caching performance. NHC can be described in three steps. First, when warming up the system (or after a significant workload change), NHC leverages classic caching to identify the current working set and load that data into the $\mathrm{D_{hi}}$; this ensures that NHC performs at least as well as classic caching. Second, after the hit rate has stabilized, NHC improves upon classic caching by sending excess load to the capacity device, $\mathrm{D_{lo}}$. This excess load has two components: read hits that are not delivering additional performance on $\mathrm{D_{hi}}$ because $\mathrm{D_{hi}}$ is already at its maximum performance, and read misses that cause unnecessary data movement between the two devices. Classic caching moves data from $\mathrm{D_{lo}}$ to $\mathrm{D_{hi}}$ when a miss occurs to improve the hit rate. However, improving hit rate is not beneficial when $\mathrm{D_{hi}}$ is already delivering its maximum performance. Therefore, NHC decreases the amount of data admissions into $\mathrm{D_{hi}}$. Using a feedback-based approach, NHC determines the excess load; it requires no knowledge of the device or the workload. Finally, if a workload change is observed, NHC returns to classic caching; if the workload never stabilizes, the algorithm degenerates to classic caching. NHC can leverage the same write-allocation policies as a classic cache (e.g., write-around or write-back).

4.3.1 Formal Definitions

To describe NHC, we introduce a few terms. We assume that the storage hierarchy is still composed of two devices, $\mathrm{D_{hi}}$ and $\mathrm{D_{lo}}$. Caching performance is determined by how the workload is distributed across those two devices. We denote the total workload over a time period δ_t as a constant W, a finite set of accesses to data items. We use w to refer to the subset of W served by $\mathrm{D_{hi}}$, and use its complement set $W - w$ for that served by $\mathrm{D_{lo}}$. We model performance in the time period δ_t as $\mathbf{P}(W, w) = \mathbf{p}_{\mathrm{hi}}(w) + \mathbf{p}_{\mathrm{lo}}(W - w)$. We make the following assumptions about the devices:

Assumption 1: Performance of a device has an upper bound. The performance of a device cannot increase after it is fully utilized. L_{hi} and L_{lo} represent the maximum possible performance that can be delivered by each device for the current workload, W. We note w_0 as the smallest subset of w such that $\mathbf{p}_{hi}(w_0) = L_{hi}$.

Assumption 2: Increasing the workload on a device does not decrease performance. This implies $\mathbf{p}_{hi}(x)$ and $\mathbf{p}_{lo}(x)$ are monotonically increasing functions. Note that HDD performance can decrease with more random requests due to more seeks, but the assumption generally holds for high-performing devices such as DRAM, NVM, and SSDs.

Assumption 3: Before reaching upper limits, $\mathbf{p}_{hi}(x)$ has a larger gradient than $\mathbf{p}_{lo}(x)$. Based on our observations from real devices, the potential performance gain of using D_{hi} is greater than that of using D_{lo}.

We define classic caching as an approach that optimizes $\mathbf{P}(W, w)$ by maximizing only $\mathbf{p}_{hi}(w)$. Classic caching attempts to maximize $\mathbf{P}(W, w)$ by finding some working set w_{max} that maximizes the hit rate of D_{hi}.

The key insight of NHC is that, when $w_0 < w_{max}$, an opportunity exists to move some portion of the workload $w_{max} - w_0$ away from D_{hi} to D_{lo}. Since $\mathbf{p}_{hi}(w_0) = \mathbf{p}_{hi}(w_{max}) = L_{hi}$, removing $w_{max} - w_0$ from D_{hi} does not decrease the performance of D_{hi} below L_{hi} and now D_{lo} can deliver some amount of performance for $w_{max} - w_0$. Thus, NHC can always perform as well or better than classic caching.

4.3.2 Architecture

As shown in Figure 4.6, classic caching can be upgraded to NHC by adding decision points to its cache controller and a *non-hierarchical cache scheduler*. The classic cache controller serves reads and writes from a user/application to the storage devices (i.e., D_{hi} and D_{lo}) and controls the contents of D_{hi} based on its replacement policy (e.g., LRU). A new cache scheduler monitors performance and controls whether classic caching is performed and where cache read hits are served. The scheduler opti-

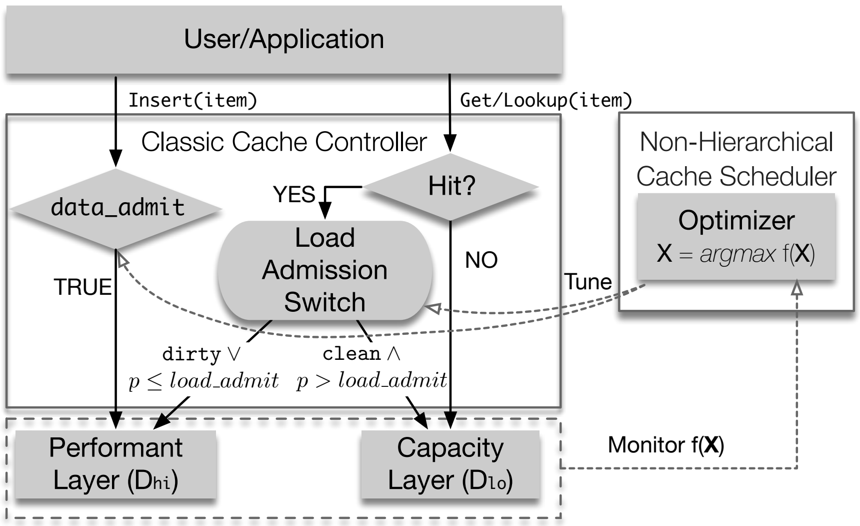


Figure 4.6: **Non-Hierarchical Caching Architecture.** *This figure shows the architecture of NHC. NHC adds decision points and a scheduler to classic caching. As before, NHC is transparent to users. Any classic caching implementation can be upgraded to be a NHC one. Note that decision points only tune cache read hits/misses.*

mizes a target performance metric that can be supplied by the end-user (e.g., ops/sec) or use device-level measurements (e.g., request latency).

The NHC scheduler performs this control with a boolean `data_admit` (`da`) and a variable `load_admit (la)`. The `da` flag controls behavior when a **read miss** occurs on $\mathrm{D}_{\mathtt{hi}}$: when `da` is set, missed data items are allocated in $\mathrm{D}_{\mathtt{hi}}$, according to the cache replacement policy; when `da` is not set, the miss is handled by $\mathrm{D}_{\mathtt{lo}}$ and not allocated in $\mathrm{D}_{\mathtt{hi}}$. Classic caching corresponds to the case where the `da` flag is true.

The `la` variable controls how **read hits** are handled and designates the percentage of read hits that should be sent to $\mathrm{D}_{\mathtt{hi}}$; when `la` is 0, all read hits are sent to $\mathrm{D}_{\mathtt{lo}}$. Specifically, for each read hit, a random number $\mathtt{R} \in [0, 1.0]$ is generated; if $\mathtt{R} <=$ `la`, the request is sent to $\mathrm{D}_{\mathtt{hi}}$; else, it is sent to $\mathrm{D}_{\mathtt{lo}}$. In classic caching, `la` is always 1.

The NHC framework works with any classic caching write-allocation policy (specified by users), which handles **write hits/misses**. NHC admits write misses into $D_{\mathtt{hi}}$ according to the policy; `da`, `la` do not control write hits/misses. With write-back, cache writes introduce dirty data in $D_{\mathtt{hi}}$ and data on $D_{\mathtt{lo}}$ can be out-of-date; in this case, NHC does not send dirty reads to $D_{\mathtt{lo}}$.

4.3.3 Cache Scheduler Algorithm

The NHC scheduler adjusts the behavior of the controller to optimize a target performance metric. As shown in Algorithm 1, the scheduler has two states: increasing the amount of data cached on $D_{\mathtt{hi}}$ to maximize hit rate, or keeping the data cached constant, while adjusting the load sent to each device.

State 1: Improve hit rate. The NHC scheduler begins by letting the cache controller perform classic caching with its default replacement policy (`da` is true and `la` is 1); during this process, the cache is warmed up and the hit rate improves as the working set is cached in $D_{\mathtt{hi}}$. The NHC scheduler monitors the hit rate of $D_{\mathtt{hi}}$ and ends this phase when the hit rate is relatively stable; at this point, the performance delivered by $D_{\mathtt{hi}}$ for the workload w_{max} is near its peak.

State 2: Adjust load between devices. After $D_{\mathtt{hi}}$ contains the working set leading to a high hit rate and performance, the NHC scheduler explores if sending some requests to $D_{\mathtt{lo}}$ increases the performance of $D_{\mathtt{lo}}$, while not decreasing the performance of $D_{\mathtt{hi}}$, i.e., the algorithm moves from w_{max} toward w_0. In this state, `da` is set to false and feedback is used to tune `la` to maximize the target performance metric. Specifically, the scheduler (Lines 6–18) modifies `la`; in each iteration, performance is measured with `la +/- step` over a time interval (e.g., 5ms see §4.4). The value of `la` is adjusted in the direction indicated by the three data points. When the current value of `la` leads to the best performance, the scheduler sticks with the current value. The value of `la` is kept in the acceptable domain of [0, 1.0] with a negative penalty function. If the scheduler finds

Algorithm 1: Non-hierarchical caching scheduler

```
cache: classic cache controller
step: the adjustment step size for load_admit
f(x): function that measures target performance metric when load_admit
  = x, the value is measured by setting load_admit = x for a time interval
1  while true do
       # State 1: Improve hit rate
2      data_admit = true,  load_admit = 1.0
3      while cache.hit_rate is not stable do
4          sleep_a_while()
5      data_admit = false,  start_hit_rate = cache.hit_rate
       # State 2: Adjust load_admit
6      while true do
7          ratio = load_admit
           # Measure f(ratio-step) and f(ratio+step)
8          max_f = Max(f(ratio-step), f(ratio), f(ratio+step))
           # Modify load_admit based on the slope
9          if f(ratio-step) == max_f then
10             load_admit = ratio - step
11         else if f(ratio+step) == max_f then
12             load_admit = ratio + step
13         else if f(ratio) == max_f then
14             load_admit = ratio
15             if load_admit == 1.0 then
16                 goto line 2 # Quit tuning if $w < w_0$
           # Check whether workload locality changes
17         if cache.hit_rate < (1-$\alpha$)start_hit_rate then
18             goto line 2
```

the optimal `la` is 1, it quits tuning and moves back to State 1; intuitively, this means NHC has moved the current workload w below w_0 and hence requires classic caching to improve the hit rate to further improve performance.

Since NHC relies on classic caching to achieve an acceptable hit rate, it restarts the optimization process when workload locality changes. The NHC scheduler monitors the cache hit rate at runtime; if the current hit rate drops, the scheduler re-enters State 1 to reconfigure the cache with the current working set. If the workload never stabilizes, NHC behaves like classic caching.

Target Performance Metrics: NHC can improve different aspects of performance. NHC can be configured to optimize metrics such as throughput, latency, tail latency, or any combination. The target metrics can also capture performance at various levels of the system (e.g., hardware, OS, or application). f is a function that measures the target metric.

Write Operations: NHC handle writes with the write-allocation policy (specified by users) in the classic cache controller. NHC does not adjust the write-allocation policy because it may be chosen for factors other than performance: endurance[128, 213], persistence, or consistency[163].

Adapting to Dynamic Workloads: NHC periodically measures the target metric (e.g., throughput) using f and optimizes it by adjusting load-admission ratios (in a way similar to gradient-descent). NHC only needs Δf to determine the optimal split of accesses. Since tuning involves only one parameter (load-admission ratio), it is cheap and converges fast. NHC can thus handle frequently-changing workloads with continual tuning.

Summary: Non-hierarchical caching optimizes classic caching to effectively use the performance of capacity devices. NHC improves on classic caching in two ways. First, NHC does not admit read misses into D_{hi} when the performance of D_{hi} is fully utilized. Second, when the performance of D_{hi} is at its peak, NHC delivers useful performance from the D_{lo} device by sending some of the requests that would have hit in D_{hi} to D_{lo} instead. By determining at run-time the appropriate load, NHC obtains useful performance from D_{lo} instead of using D_{lo} only to serve misses into D_{hi}.

4.4 Implementation

We implement non-hierarchical caching in two places: Orthus-CAS, a generic block-layer caching kernel module, and Orthus-KV, a user-level caching layer for a key-value store.

Open CAS [108] is caching software built by Intel that accelerates accesses to a slow backend block device by using a faster device as a cache. It supports different write-allocation policies such as write-around, write-through, and write-back, and uses an approximate LRU policy for eviction. Open CAS is a kernel module that we modify to leverage NHC. Orthus-CAS works with all policies supported in Open CAS.

We also implement NHC within a persistent block cache for Wisckey [168], an LSM-tree key-value store. LSM trees are a good match for Optane SSD, and have garnered significant industry interest [3, 55, 129]. Wisckey is derived from LevelDB; the primary difference is that Wisckey separates keys from values to reduce amplification. While keys remain in the LSM-tree, values are stored in a log and each key points to its corresponding value in the log. Separating keys and values also improves caching because it avoids invalidating values when compacting levels; this is similar to the approach in memcached [51] for spilling data to SSD. We integrate NHC with Wisckey's persistent block cache layer. The cache keeps hot blocks (both LSM-tree key and value blocks, 4KB in size) on the cache device using sharded-LRU. Eviction occurs in units of 64 blocks. We call this implementation Orthus-KV.

Detecting Hit-rate Stability: NHC considers the hit-rate to be stable (Algorithm 1, Lines 3-4) when it changes within 0.1% in the last 100-milliseconds. This simple heuristic works well as NHC does not require perfect hit-rate-stability detection. With intensive workloads, a higher hit-rate will only let NHC bypass more hits; with light workloads, NHC switches to classic caching quickly.

Target Performance Metrics: Our implementations support three target metrics: throughput, average latency, and tail (P99) latency, with throughput being the default. When optimizing throughput, we use the Linux block-layer statistics [46] to track device throughputs. When optimizing for latency, we track the end-to-end request latency of the caching system.

Tuning Parameters: NHC implementations must measure the target metrics and tune parameters periodically. The speed at which NHC adapts to workload changes depends on both the interval between target performance measurements and the step size. With a smaller interval, tuning converges faster. Though frequent tuning means more CPU overheads, our CPU overheads are negligible. We found the Linux block layer counters [46] are not accurate when the interval is smaller than 5 ms, so we use the smallest yet accurate interval of 5 ms. A large step size leads to faster convergence but may get sub-optimal results. NHC adjusts the load ratio by 2% in each step; we have found this gives a reasonable converging time with end results similar to smaller step sizes. We leave an adaptive step size for future work.

Implementation Overhead: We find that implementing NHC into existing caching layers is fairly straightforward and requires nominal developer effort. We added only 460 (not including cache mode registration code) and 228 LOC into Open CAS and Wisckey, respectively.

4.5 Evaluation

Our evaluation aims to answer the following questions:

- How does NHC in Orthus-CAS perform across hierarchies, write-allocation policies, and target metrics? (§4.5.1)
- How does NHC as implemented in Orthus-KV perform on static workloads? (§4.5.2)

- How does Orthus-KV handle dynamic workloads? How does it adapt to changes in load and data locality? (§4.5.3)
- How does NHC compare to previous work? (§4.5.4)

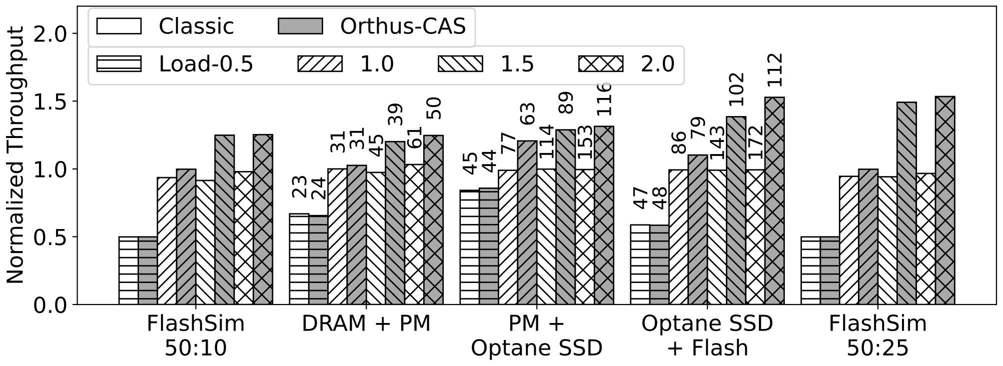
Setup. We use the following real devices: a SK Hynix DDR4 module (denoted as DRAM), an Intel Optane 128GB DC PM (PM), an Intel Optane SSD 905P (Optane), and a Samsung 970 Flash SSD (Flash). We also use FlashSim [162] to simulate flash devices with different performance characteristics.

4.5.1 Orthus-CAS

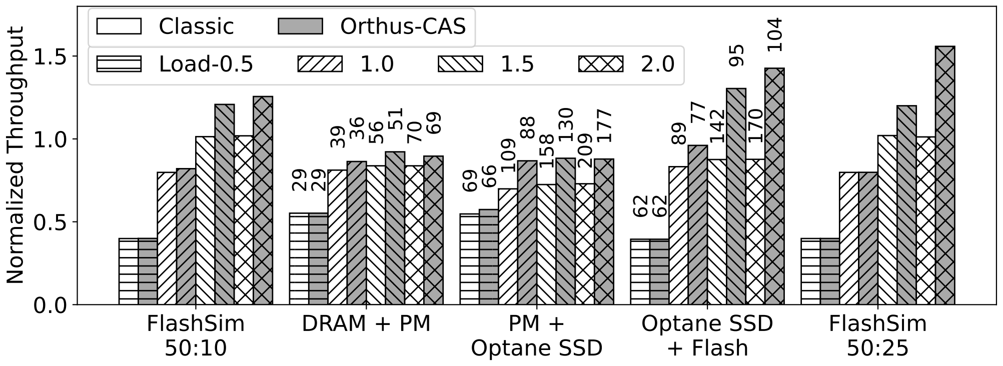
We begin by evaluating NHC as implemented in Orthus-CAS running on microbenchmarks where the workloads do not change over time. The accesses are uniformly random and 64KB (the suggested page size for Open CAS). We use 1GB of the cache device and generate workloads with different hit ratios. We report the stable performance of classic caching; for NHC, we report when its tuning stabilizes. Unless otherwise noted, we use throughput as the target function.

4.5.1.1 Storage Hierarchies

We show the normalized throughput of Open CAS and Orthus-CAS for read-only workloads with different hierarchies, amounts of load, and hit ratios in Figure 4.7. We define Load-1.0 as the minimum read load to achieve the maximum read bandwidth of the cache device; we generate Load-0.5, 1.5, and 2.0 by scaling load-1.0. We investigate hierarchies that include DRAM, PM, Optane SSD, and Flash. We also mimic hierarchies with two performance differences (50:10 and 50:25) using FlashSim; we configure FlashSim to simulate devices with maximum speeds of 50MB/s, 25MB/s, and 10MB/s. We make the following observations from the figure:



(a) 95% Hit Rate



(b) 80% Hit Rate

Figure 4.7: **Orthus-CAS on Various Hierarchies.** *This figure compares the performance of Orthus-CAS and Classic caching on various hierarchies (x axis). We use read-only workloads of varying intensity: we define Load-1.0 as the minimum read load to achieve the maximum read bandwidth of the cache device; we generate Load-0.5, 1.5, and 2.0 by scaling load-1.0. (a) and (b) show different cache hit rates. All throughputs are normalized to the maximum read bandwidth of the caching device. We show latency (μs) on top of each bar (not comparable across hierarchies).*

First, when load is light (e.g., Load-0.5), cache devices always outperform capacity devices. In this case, NHC does not bypass any load and behaves the same as classic caching.

Second, when the workload can fully utilize the cache device, Orthus-CAS improves performance. Intuitively, a higher hit ratio and load gives NHC more opportunities bypass requests and improve performance. Figure 7 confirms the intuition: with 95% hit ratio and Load-2.0, NHC obtains improvements of 21%, 32%, 54% for DRAM+PM, PM+Optane, and Optane+Flash, respectively. Such improvements are marginally reduced with an 80% hit ratio.

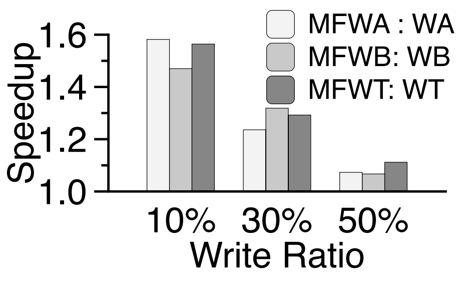
Third, among these hierarchies, Optane+Flash improves the most with Orthus-CAS since the performance difference between Optane and Flash is the smallest, followed by PM+Optane and PM+PM. Our results with FlashSim show how practitioners can predict the improvement of using NHC on their target hierarchies.

Finally, our measurements indicate that Orthus-CAS adapts to complex device characteristics. With an 80% hit ratio, classic caching does not achieve 1.0 normalized throughput on any real hierarchy because cache misses introduce additional writes to the cache device. NHC handles such complexities.

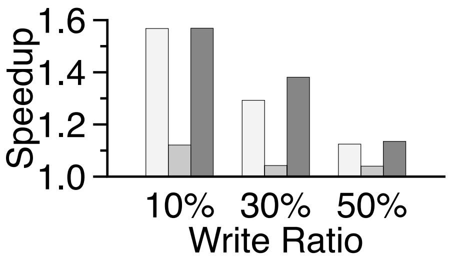
Latency Improvement. As shown in Figure 4.7, Orthus-CAS also improves average latency on all hierarchies. For instance, with Load-2.0, NHC reduces average latency by 19%, 25%, 39%, for DRAM+PM, PM+Optane, and Optane+Flash hierarchies, respectively.

4.5.1.2 Write-Allocation Policies

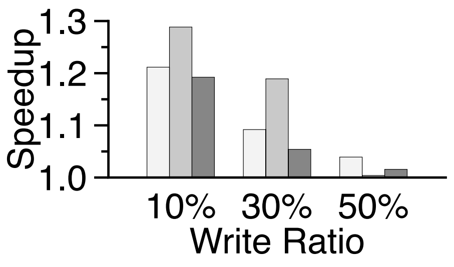
Open CAS can use a variety of write-allocation policies (write-around, write-back, and write-through) and Figure 4.8 shows that NHC improves performance relative to classic caching with each policy. The experiments vary the storage hierarchy, the write ratio, and the **dirty-read ratio**. We



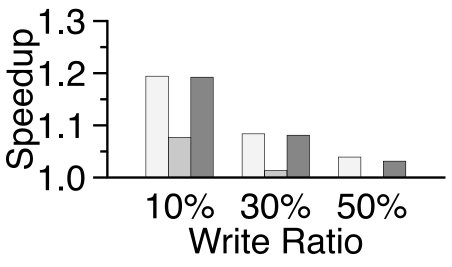
(a) Optane + Flash, 20% Dirty Reads



(b) Optane + Flash, 80% Dirty Reads



(c) PM + Optane, 20% Dirty Reads



(d) PM + Optane, 80% Dirty Reads

Figure 4.8: **Orthus-CAS with Different Write-allocation Policies.** *The figure shows Orthus-CAS overall throughput speedup (against Open CAS) with different write-allocation policies: WA, WB, and WT are write-around, write-back, and write-through. Workloads have a concurrency level of 16 and 95% hit rates.*

control the dirty-read ratio by limiting the percentage of the working set that writes can touch (e.g., if writes go to 80% of the working set, then 80% of the reads will be dirty).

NHC improves reads irrespective of write ratios. When reads or writes overload the cache device, NHC bypasses read hits, improving performance (e.g., significantly so on PM+OptaneSSD where PM writes interfere with reads dramatically). As shown in Figure 4.8, the overall improvements are dependent upon a combination of the workload write and dirty-read ratios and the write-allocation policy. NHC offers more benefits when there are fewer writes. With write-back, NHC cannot of-

Target Metric	PM + Optane			Optane + Flash		
	Throughput GB/s	Avg. lat. μs	P99 lat. μs	Throughput GB/s	Avg. lat. μs	P99 lat. μs
Open CAS	6.7	77	115	2.3	227	269
Throughput	**8.0**	**64**	147	**3.9**	**132**	289
Avg. lat.	**8.0**	**64**	143	**3.9**	**132**	285
P99 lat.	6.9	75	**106**	3.3	155	**245**

Table 4.2: **Different Target Metrics.** *The table shows Orthus-CAS performance using different target performance metrics. We use read-only workloads (concurrency level of 8, 95% hit ratio). The best result for each metric is in bold.*

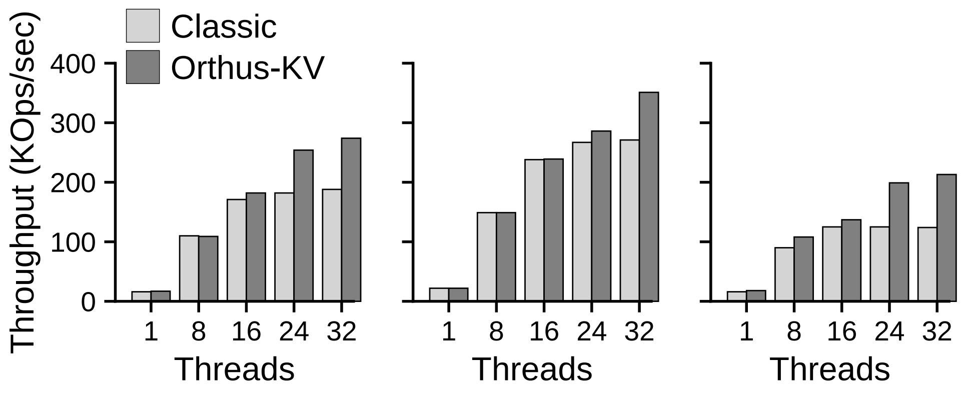
fload reads of dirty items to the capacity device and thus performs much better with fewer dirty reads. Write-around and write-through maintain consistent copies and thus Orthus-CAS offers benefits independent of the dirty-read ratio.

4.5.1.3 Target Performance Metrics

NHC can improve different performance metrics by using different measure functions f. Table 4.2 shows the performance of Open CAS and Orthus-CAS when using throughput, average latency, and tail (P99) latency as target metrics. Optimizing throughput or average latency yields similar improvements to both metrics on both hierarchies, but increases tail latency. This increase occurs because in each of these storage hierarchies, the performance device has much better tail latency than the capacity device; thus classic caching defaults to appropriate behavior. When NHC is configured with P99 latency as the target metric, Orthus-CAS has better tail latency than Open CAS and than it does with other targets.

4.5.2 Orthus-KV: Static Workloads

We use Orthus-KV, the NHC implementation in Wisckey, to show the benefits for real applications. Caching in Wisckey uses write-around, due to the LSM-tree's log-structured writes. In these experiments we focus on



(a) V:1KB theta:0.6 (b) V:1KB theta:0.8 (c) V:16KB theta:0.8

Figure 4.9: **Orthus-KV, YCSB-C Performance.** *This figure shows Orthus-KV performance with YCSB-C benchmark. YCSB-C workload has 100% reads. We use 16B keys and 1KB or 16KB values. Accesses follow a Zipfian distribution (theta).*

Optane+Flash since it is often used for key-value stores[55, 129]. We set the caching layer to 33 GB, 1/3 of the 100 GB dataset. We use `cgroup` to limit the OS page cache to 1 GB to focus on caching in the storage system instead of caching in main memory.

Our initial evaluation uses the YCSB workloads [117]. Most YCSB workloads are constant: their load does not change and they have a stable key popularity distribution (i.e., Zipfian). These workloads cover different read/write ratios (e.g., YCSB-C: 100% reads, YCSB-A: 50% reads and 50% updates), as well as various operations (e.g., YCSB-E involves 95% range queries and 5% inserting new keys, while Workload-F has 50% read-modify-writes). We evaluate YCSB-D as a dynamic workload (§4.5.3).

Gets: Figure 4.9 compares the throughput of Wisckey and Orthus-KV for three YCSB-C workloads and different amounts of concurrency. Orthus-KV achieves equivalent or higher throughput than Wisckey for all workloads. Orthus-KV significantly improves throughput at high load

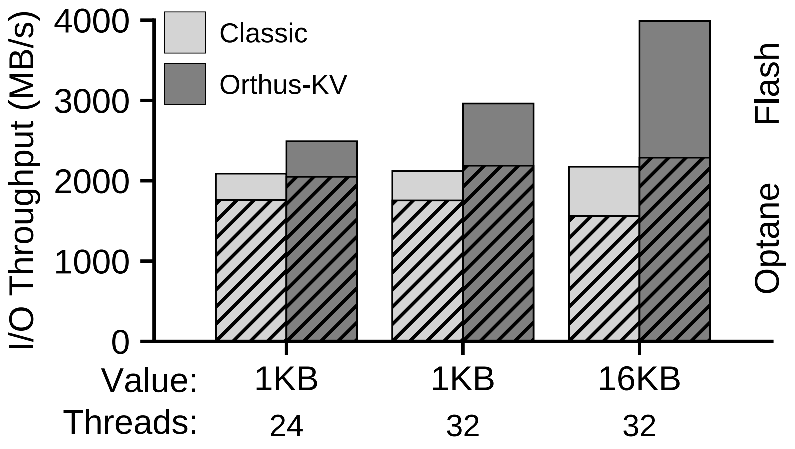


Figure 4.10: **Orthus-KV Bandwidth Breakdown, YCSB-C.** *This figure compares Optane/Flash SSD read bandwidth breakdown during YCSB-C workloads for Orthus-KV (dark grey bars) and Classic Caching (light grey bars).*

levels: with 32 threads, 46%, 30%, and 71% higher throughput for the three workloads. When load is high enough to saturate Optane, the relative benefits of Orthus-KV depend on how much it can avoid unnecessary data movement and perform better load distribution. Figure 4.10 illustrates these two benefits with the read bandwidth delivered by each device. First, classic caching suffers from unnecessary data admissions into Optane: its effective Optane read bandwidth never reaches the peak (2.3GB/s). NHC avoids this wasteful data movement. Second, classic caching never delivers more than the maximum Optane bandwidth to the application. In contrast, NHC improves the performance out of the hierarchy by distributing some cache hits to the Flash SSD.

Updates, Inserts, and Range Queries: Figure 4.11 shows Orthus-KV handles a range of operations (gets, updates, inserts, and range queries) and always performs at least as well as Wisckey. NHC improves all YCSB workloads, with greater benefits with more get operations.

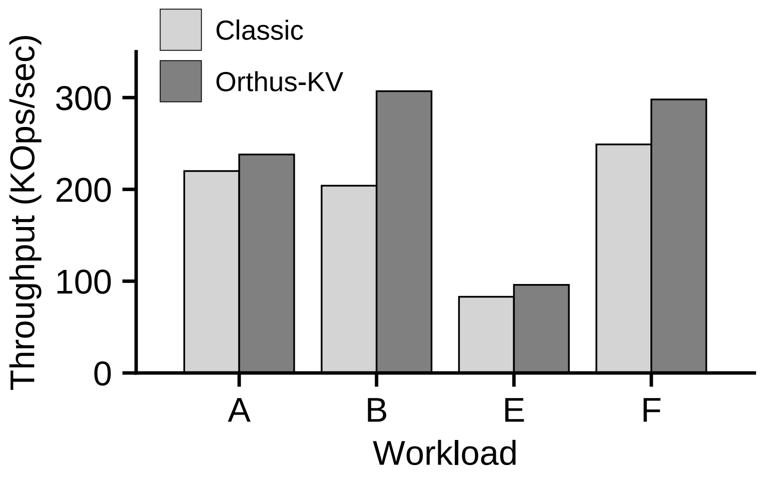


Figure 4.11: **Other YCSB Workloads.** *This figure shows Orthus-KV performance for YCSB A,B,E,F workloads. We use 16B keys, 1KB values, 32 threads and theta = 0.6.*

Latency Improvement: With throughput as its target, Orthus-KV reduces YCSB average latency by up to 42%. For YCSB-C (32 threads, 0.8 theta), Orthus-KV provides 30% and 38% lower latency for 1KB and 16KB values, respectively.

CPU Overhead: Orthus-KV increases CPU utilization slightly (0-2% for 24 threads) due to a few additional counters that track caching behavior and device performance over time.

4.5.3 Orthus-KV: Dynamic Workloads

We evaluate NHC for dynamic workloads using the same experimental setting as §4.5.2. We explore how Orthus-KV handles time-varying workloads and dynamic working sets.

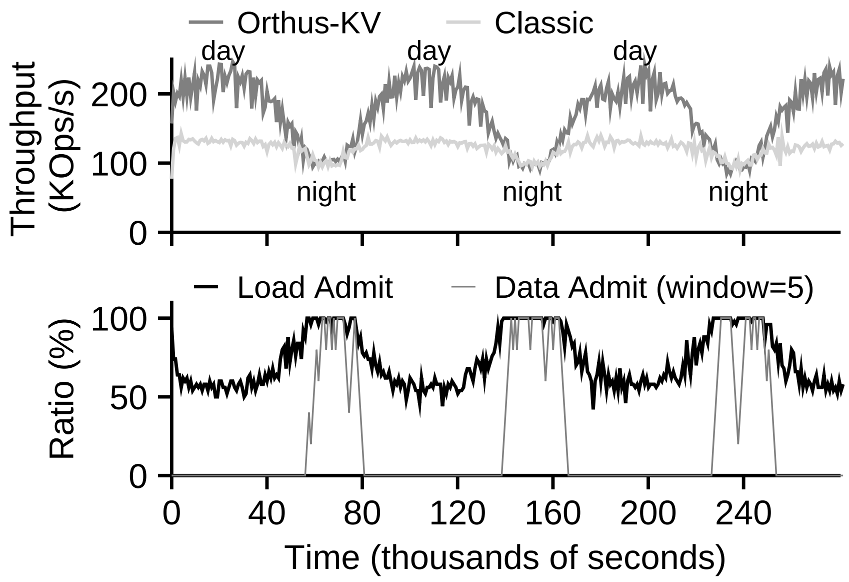


Figure 4.12: **Orthus-KV with Dynamic ZippyDB Workloads.** *This figure shows the throughput of Orthus-KV and classic caching (top graphs), as well as load/data admit ratio over time in Orthus-KV (bottom graphs). Because data admit is 0 or 1, we show a fractional windowed sum of its value over 5 time steps for readability. We replay the ZippyDB benchmark on a single machine. The average value size is 16KB; the number of key ranges is 6. We use 32 threads for the maximum load and adjust the number of threads dynamically according to the diurnal load model. We speed up the two-day workload by 1000×.*

4.5.3.1 Dynamic Load

We evaluate how well NHC handles load changes with the Facebook ZippyDB benchmark [107]. ZippyDB is a distributed key-value store built on RocksDB and used by Facebook. The ZippyDB benchmark generates key-value operations according to realistic trace statistics: 85% gets, 14% puts, 1% scans following a hot range-based model; the request rate models the diurnal load sent to ZippyDB servers. We note that the access patterns (e.g., read sizes) of the ZippyDB benchmark vary significantly as their value sizes range from bytes to MBs. We speed up the replay of Zippydb requests by 1000 to stress the storage system and to better evaluate NHC's reactions to changes in load.

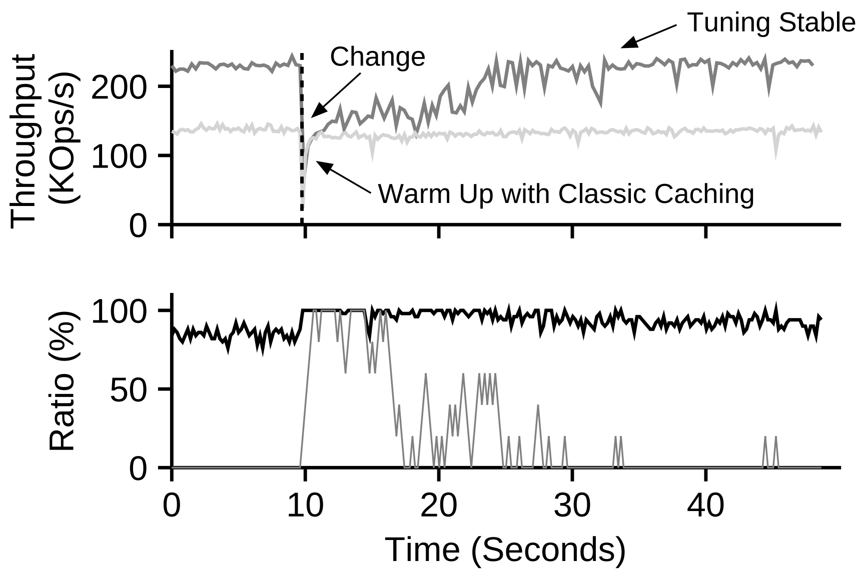


Figure 4.13: **Orthus-KV with a Sudden Working Set Change in Workload.** *This figure shows throughput of Orthus-KV and classic caching (top graphs), and load/data admit ratio over time in Orthus-KB (bottom graphs). We use the workload similar to YCSB-C 16KB value, 32 threads, but with two different working sets before and after 10s.*

As shown in Figure 4.12 (top), Orthus-KV outperforms Wisckey during the day by up to 100%, but performs similarly when the load is lower at night. Figure 4.12 (bottom) shows how Orthus-KV adjusts data and load admit ratios. During the night, both are around 100%; Orthus-KV occasionally adjusts the load admit ratio when the hit rate is stable, but quickly returns to classic caching after finding no improvements. During the day, Orthus-KV keeps the data admit ratio at 0 and adjusts the load admit ratio to adapt to the dynamic load.

4.5.3.2 Dynamic Data Locality

We demonstrate that NHC reacts well to abrupt changes in the working set in Figure 4.13. The experiments base on YCSB-C, beginning with one working set (Zipfian theta=0.8, hot spot at beginning of the key space),

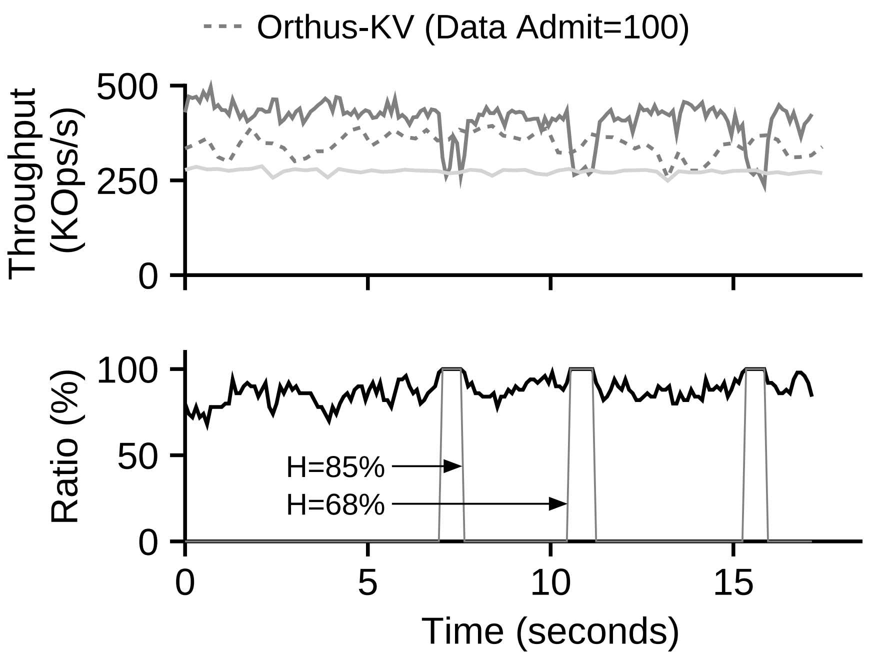


Figure 4.14: **Orthus-KV with a Gradual Working Set Change in Workload.** *This figure shows throughput of Orthus-KV and classic caching (top graphs), and load/data admit ratio over time in Orthus-KB (bottom graphs). We use YCSB-D with 16B keys, 1KB values, 32 threads. We also show throughput of a modified Orthus-KV that always performs data admit (dotted line).*

and then changing at time 10s (a hot spot at the end of the key space). The graph shows that when the working set changes (time=10s), Orthus-KV quickly detects the change in hit rate and switches to classic caching: the load and data admit ratios increase to 1.0. After the hit rate begins to stabilize (time=11s), Orthus-KV tunes the load admit ratio. Initially (11s-28s), because the hit rate is still not high enough,Orthus-KV often identifies 1.0 as the best load admit and returns to classic caching with data movement. Approximately 20s after the workload change, the hit rate stabilizes and Orthus-KV reaches steady-state performance that is 60% higher than classic caching.

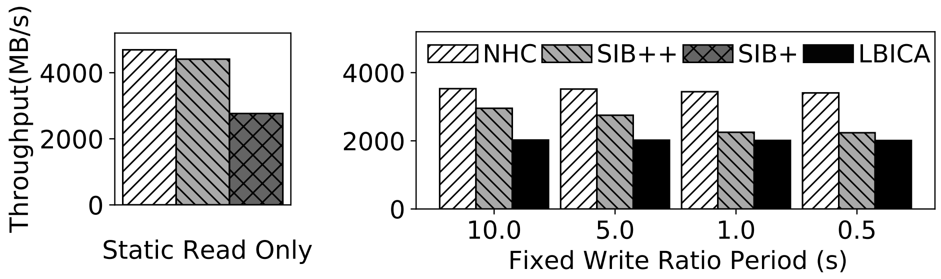
Finally, we show that NHC can outperform classic caching even when the working set changes gradually. Figure 4.14 shows Orthus-KV's performance on YCSB-D (95% reads, 5% inserts), where locality shifts over time as reads are performed on recently-inserted values. Due to the locality changes and not admitting new data to the cache, the hit rate in Orthus-KV decreases over time, until NHC identifies that 1.0 is the best load admit rate. Then Orthus-KV returns to classic caching and raises the hit rate. Once the hit rate restabilizes, the cycle resumes with Orthus-KV adjusting the load admit rate.

We also explore the alternative approach of always performing data admission while tuning the load admit rate in Figure 4.14. As shown, this alternative maintains a stable hit rate, avoiding abrupt phases of admitting new data; this always performs better than classic caching but its peak performance does not reach that of the default Orthus-KV. Our results illustrate the interesting tradeoff between avoiding unnecessary data movement and maintaining a stable hit rate for dynamically changing workloads.

4.5.4 Comparisons with Prior Approaches

We now show that NHC significantly outperforms two other approaches that have been suggested for utilizing the performance of a capacity device: SIB [160] and LBICA [80]. SIB targets HDFS clusters with many SSDs and HDDs, in which case the aggregate HDD throughput is non-trivial: SIB uses SSDs as a write buffer (does not admit any read miss), and proposes using the HDDs for handling extra read traffic. LBICA determines when a performance layer is under "burst load" at which point it will not allocate new data to the performance layer; unlike NHC, LBICA does not redirect any read hits.

To compare NHC against SIB and LBICA, we have implemented these approaches in Open CAS. To make SIB suitable for general-purpose caching environments, we have improved it in two ways. First, SIB oper-



(a) Static Workload

(b) Dynamic Workloads

Figure 4.15: **NHC vs. SIB and LBICA.** *This figure compares NHC with two related works SIB and LBICA. (a) uses a static read-only workload. (b) uses dynamic workloads; the write-ratio is fixed in each period (e.g., 10s), but changes (randomly between 0% to 50%) across periods. We use workloads with parallelism/concurrency 16, 95% hits, runtime 100s on Optane+Flash hierarchy.*

ates on a per-process granularity instead of per-request: the traffic from some processes is not allowed to use the SSD cache; we altered SIB so that it adjusts load per-request (SIB+). Second, we modified SIB so that it admits read misses into the cache (SIB++).

Using the experimental setup from §4.5.1 on Optane+Flash, we start with a read-only workload in Figure 4.15.a. SIB+ does not perform well because it does not admit read misses into Optane. SIB++ performs better, but suffers when the workload changes as in Figure 4.15.b; in these workloads, the amount of write traffic is changed every period, for periods between 10 and 0.5 seconds. SIB cannot handle dynamic workloads because SIB has two phases; in its training phase, SIB learns the maximum performance of the caching device for the current workload; in the inference phase, SIB judges whether the caching device is saturated and, if it is, bypasses some processes (requests). As we have shown, the maximum performance of modern devices depends on many workload parameters: read-write ratios, load, and access patterns. Thus, if the workload changes in any way, SIB must either relearn the target maximum performance or use an inaccurate target. In our experiments, as the duration of each phase decreases, the performance of SIB decreases dramatically. Unlike SIB, NHC uses a simple, continuous feedback-based tuning approach and thus converges rapidly and adapts to dynamic workloads well. Finally, LBICA performs poorly because it does not redirect any read hits to use the capacity device; it simply does not allocate more data to the performance device when it is overloaded.

4.6 Conclusions

In this chapter, we show how emerging PM devices have strong implications for caching in modern hierarchies. We show that the decades-old caching principle of maximizing hit rates ignores capacity layer performance; however, in modern hierarchies, capacity layers (such as PM DIMM) are performant and can account for a significant portion of overall hierarchy performance (cache layer + capacity layer). As a result, in modern hierarchies, this principle becomes insufficient. We introduced non-hierarchical caching (NHC), an enhanced caching approach to extract peak performance from modern hierarchies. Compared to classic caching, NHC has three new components: i) read around mechanism, ii) admission rejection mechanism, and iii) a feedback-based offloading policy. The first two components enable NHC to offload accesses to capacity layers when the cache device is already saturated, and exploit the capacity layer performance. The third component help determine how much to offload during runtime in response to various device and workload characteristics in the hierarchy. Through experiments, we demonstrated the benefits of NHC on a wide range of devices, cache configurations, and workloads. We believe NHC can serve as a better foundation to manage storage hierarchies.

5 Evolving Sharing Mechanisms for PM

In the last part of the thesis, we address the question of how PM devices can be shared across multiple tenants. We analyze the problems of existing sharing mechanisms (designed for DRAM/block devices) on PM, and then propose new mechanisms to overcome the difficulties of sharing PM. We focus on a detailed setup of multi-tenant memory-based look-aside key-value caches (such as memcached [52]). This chapter is based on the paper, *NyxCache: Flexible and Efficient Multi-tenant Persistent Memory Caching*, published in FAST 2022 [231].

To analyze existing sharing approaches for PM, we first summarize the basic mechanisms (e.g., resource usage accounting, interference analysis) used to achieve diverse sharing objectives (e.g., QoS-aware, proportional sharing). This is an important first step because, as we have discovered, there are numerous sharing objectives in various sharing configurations; studying all of these objectives would be impossible. Then, we examine each sharing mechanism's issues on PM. Our analysis shows that existing sharing mechanisms designed for DRAM/block devices do not readily translate to PM due to PM's unique characteristics such as 256B access granularity, asymmetric read/write performance, and severe and unfair interference between reads and writes.

To address this issue, we introduce NyxCache (Nyx), a standalone lightweight and flexible PM access regulation framework for multi-tenant key-value caches that is optimized for today's PM without special hardware support. Nyx's central contribution is a set of software mechanisms designed for PM to extract the information required to flexibly enforce popular sharing policies. Nyx provides new mechanisms to efficiently i) regulate PM accesses, ii) obtain a client's PM resource usage, and iii) analyze inter-client interferences for PM. We then describe how we use these new mechanisms to easily and efficiently support sharing policies such as resource limiting, QoS, fair slowdown, and proportional sharing. Through various experimental studies, we demonstrate the efficacy of each sharing policy.

In the following, we first evaluate previous multi-tenant caches and their limits for PM (§5.1); discuss the Nyx design (§5.2); evaluate overheads of Nyx's mechanisms and the effectiveness of its policies (§5.3); discuss potential extensions (§5.4); and conclude (§5.5).

5.1 Background and Motivation

We provide background on the sharing policies provided by many in-memory multi-tenant key-value caches and the mechanisms needed to implement those policies. We explain why previous approach for providing control and information on DRAM or block devices do not work well on PM.

5.1.1 Sharing Policies for Multi-Tenant Caches

In-memory key-value caches such as memcached [52], Redis [66], and Pelikan [61] are an essential part of web infrastructure for many real-time and batch applications [10, 239]. Before accessing data from slow backend-storage or compute nodes, applications first check an in-memory cache server. In production environments, cache servers are usually multi-tenant: many cache instances are consolidated on a single server to improve utilization and simplify management and scaling [187]. In a multi-tenant cache, requests are routed to the cache instance of the corresponding tenant. For example, large companies such as Facebook [187] and Twitter [239] maintain hundreds of large-memory dedicated servers that host thousands of cache instances. Smaller companies use caching-as-a-service providers such as ElastiCache [7], Redis [65] and Memcachier [54]. In this chapter, we focus on managing an individual multi-tenant cache server.

Giving competing clients, enforcing performance and sharing goals is critical in multi-tenant caching. Different industrial and research multi-tenant systems have provided different objectives; we focus on the following four.

Resource Limiting: A common objective when clients pay for resources is to guarantee that each client cannot exceed some amount of usage such as bandwidth, ops/sec, or number of resources [6, 27]. For example, Google Cloud memcache limits operations according to a pricing tier, such as "Up to 10k reads or 5k writes (exclusive) per sec per GB" [27]. Multiple resources can be limited simultaneously, e.g., Amazon ElastiCache [6] charges for both memory and vCPUs.

There are two requirements for a multi-tenant cache to enforce per-client resource limits. First, the system must accurately determine the amount of resource each client is using; we refer to this as *resource usage estimation*. Second, the system must reschedule or throttle requests of each client if they exceed this limit, which we call *request regulation*. Below (§5.1.2), we describe how previous multi-tenant caches have provided request regulation and resource usage estimation, and why these previous approaches are not sufficient for PM.

Quality-of-Service: A multi-tenant system may ensure that each client's performance goals (throughput, latency, or tail latency) are met regardless of other co-located clients, as in Twitter [62] and Microsoft [214]. This objective is useful for latency-critical clients that must meet service-level objectives (SLOs). For example, production caches at Twitter provide a p999 latency of <5 milliseconds [62].

Providing QoS requires knowledge of whether each client is meeting its goals at run-time. When the system observes that one client is not meeting its performance guarantee, interfering clients are identified and limited [111, 135, 176] (e.g., with *request regulation*). Identifying the client causing the most harm is usually straightforward and based on simple bandwidth [135] for DRAM-based caches, but not for PM. A new technique involving *interference estimation* is required on PM to determine how the workloads compose.

In addition to run-time support, guaranteeing QoS requires admission control and space allocation. Admission control must be performed on newly arriving clients to ensure that the system has sufficient resources and that the new client will not interfere with existing clients [120, 121, 179]. Space allocation across cache instances must be performed to provide a specified hit ratio for each client to ensure each can meet its goals. Previous research has focused on this challenge. For example, Microsoft [214] allocates space to meet QoS bandwidth targets, and Robinhood [96] to minimize tail latency. Admission control and space allocation are mostly orthogonal to the new challenges introduced by PM and are not our focus.

Fair Slowdown: Multi-tenant systems in more cooperative environments may ensure that all clients are slowed down by the same amount. Formally, these approaches minimize the ratio of the maximum slowdown to the minimum slowdown [127, 215]. In web cache settings, application requests may fan out, in which case the cache access with the longest latency determines overall latency [96, 187]; thus, balancing slowdown benefits overall request latency.

Enforcing fair slowdown requires knowledge of each cache's slowdown at runtime. The system must monitor each cache's current perfor-

	Resource Limit	**Quality of Service**	**Fair Slowdown**	**Proportional Resource Allocation**
Request Regulation	✓	✓	✓	✓
Resource Usage	✓			✓
Interference		✓		*
Application Slowdown			✓	✓

Table 5.1: **Control and Information Needed.** *This figure summarizes the control and information required to implement popular sharing goals (Resource Limit, etc.). ✓indicates control or information is required by the policy. * indicates optional.*

mance when sharing the server with others and know its performance if run alone. A technique for *slowdown estimation* is required. Furthermore, to equalize slowdowns of different caches, caches with small slowdown should be further limited and caches with larger slowdowns should be less limited (e.g., with *request regulation*).

Proportional Resource Allocation: Finally, a multi-tenant system may incent clients to share resources by guaranteeing that each of N clients performs within 1/N-th of its stand-alone performance. This guarantee can be generalized to give each client a different proportional share. Idle resources may be redistributed across clients, such that some obtain more than their guarantee. For example, FairRide [199] ensures proportional cache space allocation.

To guarantee proportional allocation, a multi-tenant cache must meet three requirements. The system must perform *request regulation* and *resource usage estimation* to guarantee that each client does not consume more than its allocation. When assigning idle resources to clients, the system must validate that the additional resource usage does not interfere with others; therefore, the system must track each client's slowdown (i.e., with *slowdown estimation*) and stop idle resource re-allocation before it severely impacts some clients.

In summary, for a multi-tenant cache to provide the above policies, it

	Metric	Load	No-Prefetch	NT-Load	Store	Store+clwb	NT-Store
256B	GB/s	1.59	1.53	0.29	1.12	0.52	3.73
	us	0.49	0.52	0.84	0.38	0.47	0.08
4KB	GB/s	4.08	2.92	2.24	1.03	1.50	3.44
	us	1.22	1.69	1.84	4.14	2.71	1.22

Table 5.2: **PM Load/Store Performance.** *This table summarizes the throughput and latency of single thread random 256B and 4KB load/store operations (on 2× Optane DC PM). No-Prefetch: the CPU's prefetching is turned off (for DRAM/PM); NT: non-temporal operations that bypass the CPU cache.*

must control resource usage of each cache instance and obtain information about resources and application performance. Table 5.1 summarizes the needed control and information for each policy.

5.1.2 Challenges of PM Cache Sharing

Persistent memory is an appealing building block for key-value caches. After presenting PM background, we describe the challenges of using PM for multi-tenant caching.

5.1.2.1 Persistent Memory Characteristics

PM is becoming a reality in products and research prototypes. For example, Intel Optane DC PM [37] is a popularly available device; there are also research prototypes [109, 169, 227]. In this chapter, we use PM to refer to Optane DC PM. PM performance is similar to DRAM but can deliver extremely large capacity at low cost [37, 38]. PM is significantly faster than NAND Flash and is byte-addressable. PM is directly connected to the memory bus and, when configured in App Direct Mode, can be accessed using loads and stores. Different CPU caching options exist for PM access: loads and stores with CPU caching and prefetching; loads and stores with prefetching disabled (for both PM and DRAM); non-temporal (NT) operations that bypass the CPU cache entirely [238].

Table 5.2 summarizes the bandwidth and latency of Optane DC PM for a workload relevant to key-value caches: random 256B and 4KB loads and stores. As shown, for loads, regular loads perform best: CPU cache prefetching is essential for hiding PM latency and increasing throughput. For stores on a random workload, NT-stores that bypass the CPU cache have much better performance. Thus, we use in-PM key-value caches optimized to use regular loads and NT-stores.

PM has unique characteristics that impact multi-tenant caching. For instance, as previously identified, PM exhibits asymmetric read vs. write performance [151], especially efficient access for specific sizes (e.g., 256B) [238], and severe and unfair interference across reads and writes [189]. As we will describe, these characteristics deeply impact the ability to perform request regulation and to estimate resource usage, interference, and application slowdown.

5.1.2.2 Request Regulation

Previous approaches for request regulation have been designed for both DRAM and for block I/O. However, none of these approaches are suitable for PM.

Existing techniques for regulating memory requests have adjusted the number of cores dedicated to an application [135], used clock modulation (DVFS) [196], and Intel Memory Bandwidth Allocation (MBA) [35]. In multi-tenant caching, reducing the number of cores is not suitable because a cache instance is often allotted only a single core [6]. Intel MBA manages last-level cache (LLC) misses from each core to limit memory traffic, but does not distinguish between misses to PM and DRAM [34] and so cannot restrict PM accesses without also slowing down DRAM. Furthermore, Intel MBA does not have access to accurate information about resource usage, interference, and application slowdown, as we will discuss. Likewise, adjusting CPU frequency has an effect on all instructions; Oh et al. [189] demonstrated the ineffectiveness of CPU frequency scaling on regulating PM traffic.

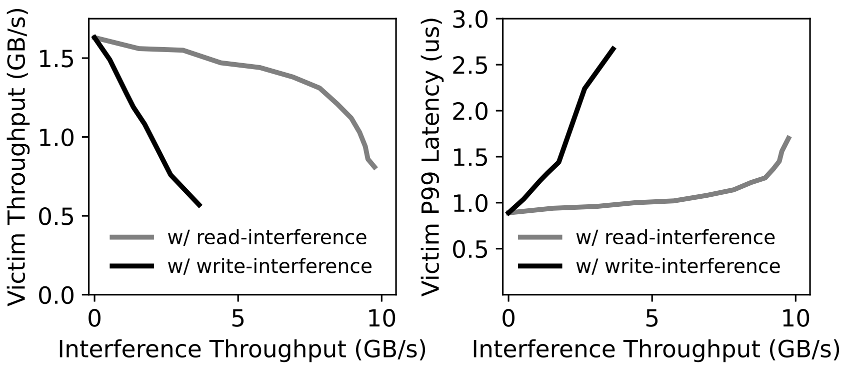
I/O requests have been regulated via software with block-layer I/O scheduling [43], which is not suitable for PM for two reasons. First, the block abstraction would add significant read/write amplification for byte-addressable PM. Second, scheduling requests with merging, reordering, and other synchronization would add unacceptable overhead to otherwise low-latency PM accesses [90].

5.1.2.3 Resource Usage Estimation

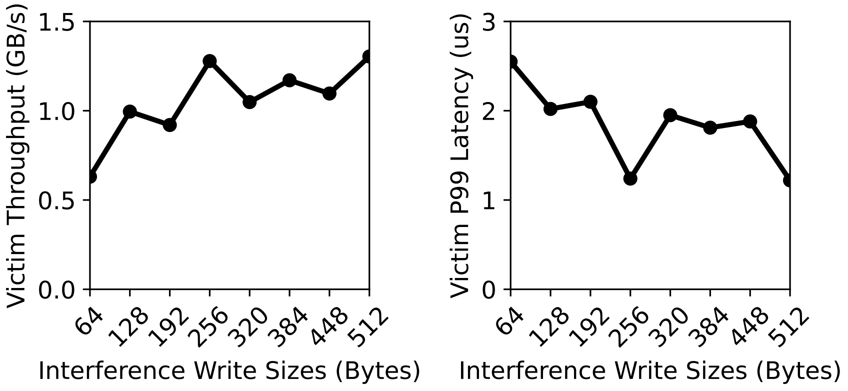
Previous techniques for estimating the memory or I/O usage of clients do not work well for PM. We describe the problems with previous software approaches for tracking I/O usage and with hardware approaches for DRAM.

As discussed above, CPU cache prefetching is required for PM to deliver high bandwidth and low latency. However, when estimating block I/O traffic in software [12, 118, 241], extra PM accesses caused by prefetching are not observed. Running an experiment with 1KB random loads, we found that software-level tracking accounted for only 60% of actual memory traffic, leading to inaccurate resource-usage estimation.

Accounting on DRAM uses hardware counters to track L3 cache line misses to the memory controller per core. While hardware counters accurately measure prefetching, they do not account for the difference between cache line size and PM access granularity, which is needed for PM accounting. Because PM has a 256B minimum access granularity, a 64B load (a single L3 cache line) utilizes the same amount of PM resources as a 256B load (four L3 cache lines). Thus, four cache line accesses can result one to four PMEM accesses. Previous systems for resource estimation have often used bandwidth consumption as a proxy for resource usage [135, 176, 243], but this is not appropriate for PM where operation



(a) Read vs. Write Interferences



(b) Interferences Related to Access Sizes

Figure 5.1: **PM Load Performance with Various Interferences.** *This figure shows the throughput of a victim workload (single thread 256B loads) with various interferences. (a) shows the victim throughput and tail latency when colocated with varying amounts of read and write interferences. (b) shows the victim performance when colocated with 1GB/s store traffic of varying access sizes (range from 64B to 512B with step of 64B).*

cost is affected by access size and is different for reads versus writes.

Unfortunately, current hardware counters in PM are also not sufficient; existing PM counters are at the DIMM media-level and do not track per-client or per-core usage [45, 189].

5.1.2.4 Interference Estimation

In memory-based approaches, interference caused by a particular client was assumed to be related to memory bandwidth. For example, Caladan [135] identifies the client with the highest number of LLC misses, which corresponds directly to the client with the highest memory bandwidth. This simplification does not work for PM, as PM interference depends on both volume and pattern of traffic.

Specifically, on PM, write-intensive clients generate greater interference than read-intensive clients with the same bandwidth, as shown in Figure 5.1.a. For example, on a read-intensive client, a competing 1GB/s write causes the same throughput and tail latency interference as a competing 8GB/s read. As shown in Figure 5.1.b, smaller accesses (64B) can cause more interference than larger accesses (256B). Since PM has a minimum granularity of 256B, a 64B access is amplified into 256B on the device; thus, at the same bandwidth, 64B accesses generate significantly more interference than 256B accesses. In short, the bandwidth of a competing client is not a good estimation of interference in PM, unlike DRAM.

5.1.2.5 Application Slowdown Estimation

Numerous efforts have estimated slowdown for DRAM and Flash-based systems; however, all require specialized device support. For example, FST [127] requires in-DRAM bank conflict counters that are updated with each memory access; MISE [216] and ASM [215] require the DRAM controller to assign priorities to application requests. FLIN [217] changes the Flash controller to track and rearrange each flash transaction. Although application slowdown is not inherently different on PM than DRAM or I/O, previous approaches require special hardware which is not available on PM.

Summary: Multi-tenant PM caching demands new methods for regulating PM accesses and extracting PM resource usage, interference information, and application slowdown.

5.2 NyxCache Design

Given that existing multi-tenant cache servers cannot handle PM, we introduce NyxCache (Nyx). Nyx provides mechanisms for control (e.g., request throttling) and information estimation on PM (e.g., resource usage, interference, and application slowdown), and supports a range of sharing policies (e.g., resource limiting, quality-of-service, fair slowdown, and proportional resource usage). We describe the overall architecture of Nyx, present our design goals, describe how Nyx provides these mechanisms and policies.

5.2.1 Architecture

As shown in Figure 5.2, Nyx provides a multi-tenant in-PM caching framework. Each PM server running Nyx may contain any number of cache instances (e.g., memcached, Pelikan, Redis). Thousands of users may send requests (e.g., set/get) to their associated cache instance. When cache space is exhausted, a cache instance can use any eviction strategy (e.g., FIFO, LRU, and LFU). As in other look-aside caches, users explicitly write desired data into the cache; Nyx does not fetch data from remote storage on a cache miss.

Nyx can be configured with different sharing policies and parameters (e.g., a resource limit, latency target, or proportional weight). Administrators can implement new policies using the control and information mechanisms provided by Nyx. At runtime, Nyx enforces the desired sharing policy. Based on information Nyx acquires about per-instance

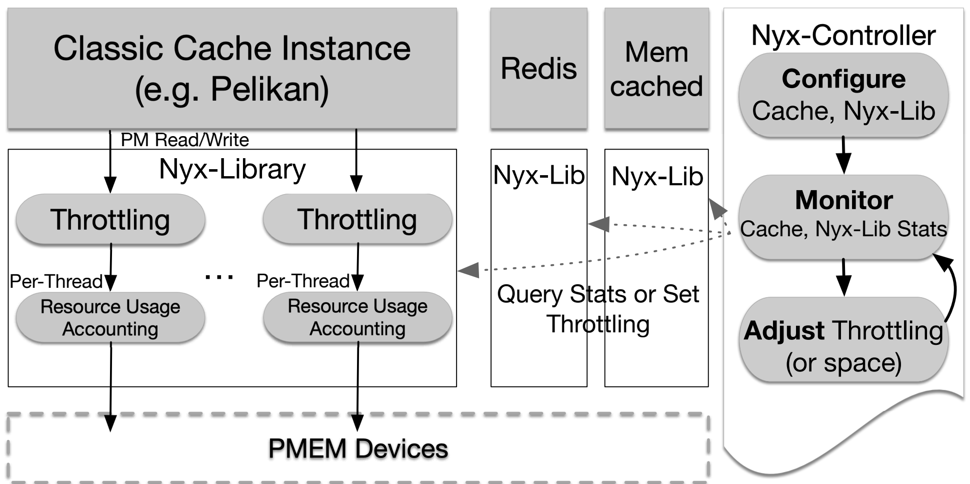


Figure 5.2: **NyxCache Architecture.** *This figure presents Nyx architecture. Nyx implements throttling and resource usage accounting for each cache instance, and enforces sharing policies across cache instances. Nyx contains two major components: 1) a Nyx Library for each instance, and 2) a centralized Nyx Controller.*

resource usage and performance, the Nyx controller dynamically adjusts the throttling and space allocated to instances.

Nyx has two requirements for cache instances. First, each cache instance must report application-level performance metrics such as throughput and tail latency; most systems have this capability or can be extended [53]. Second, the instances must be integrated with a trusted Nyx-library. When a cache instance reads/writes from/to PM, it must use Nyx library APIs (e.g., read(dest, src), write(dest, src)). For each PM access, the Nyx library throttles access, tracks PM usage, and performs the actual access. The library uses a separate thread to communicate with the Nyx controller. The controller interacts with the library to query statistics and to set configuration, space, and throttling values. Nyx leverages techniques from previous multi-tenant in-memory caches for basic sharing functionality such as admission control and space allocation. As of now, Nyx only manages cache instances on a single NUMA node that share PM (and all PM accesses are local); multiple Nyx can be used to manage multiple NUMA nodes. We leave NUMA-aware management for future work.

5.2.2 Design goals

Nyx has the following goals. (i) **Lightweight:** Performance is critical for in-PM caching; thus the cost of adding control and acquiring information must be low relative to the cost of accessing PM. (ii) **Flexible Sharing Policies:** Different sharing policies may be required by administrators for different scenarios. Thus, Nyx can be configured with several policies based on a common set of simple mechanisms. (iii) **No Special Hardware:** Previous work has assumed smart resources (e.g. Flash, DRAM) that provide configurable control and information [186, 217, 245, 247]. Nyx handles current devices with existing hardware interfaces. (iv) **Minimal Assumptions:** Storage devices are continuously evolving, with new generations having new performance characteristics. Therefore, Nyx does not assume a particular performance model for all PM devices (e.g., the interference for different operations).

5.2.3 Nyx Mechanisms

Nyx contains low-level mechanisms that enable higher-level sharing policies to be implemented easily. Since request regulation, estimation of resource usage, interference, and application slowdown are changed significantly by PM, we describe these Nyx mechanisms in detail. Access control and space allocation are largely independent of PM and not the focus of this chapter; Nyx borrows these techniques from previous systems [96, 116, 179, 199, 214].

PM Access Regulation: To minimize the overhead of regulating requests to PM, Nyx adheres to the basic principle used by previous techniques for DRAM regulation: throttle requests in a coarse-grained manner without reordering or prioritizing. To mimic the behavior of Intel MBA, Nyx implements simple throttling by delaying PM accesses at user-level.

Our current implementation adds delays in units of 10ns with a simple computation-based busy loop. In some cases PM operations may need to be delayed indefinitely (e.g., when a resource limit is reached); in this case, PM operations are stalled until the Nyx controller sets the delay to a finite value.

Resource Usage Estimation: Nyx must determine how much PM resource each cache instance is using. As described in Section 5.1, for PM the number of transferred bytes is not a good estimate of resource usage; on PM, each operation type (e.g., read or write) and access pattern (e.g., request size) consumes a different amount of the resource and has a different maximum operations per second. Therefore, Nyx determines the utilization of PM as a function of the current IOPS of each operation type relative to the maximum IOPS for that operation type. For example, if the maximum IOPS of pattern A is $MaxIOPS_A$, then the cost of each operation of pattern A is $1/MaxIOPS_A$. If the maximum IOPS of pattern B is $1/N \times MaxIOPS_A$, then each B operation consumes N times more PM than an A operation and has N times the cost. The IOPS cost model accurately captures that writes are more expensive than reads, and the dependency on request size.

Nyx determines the MaxIOPS of each access pattern through profiling, performed once per PM server. The profiler measures IOPS for random read and write operations between 64B and 4KB (in steps of 64B). Because prefetching occurs during profiling, the measured MaxIOPS accurately represents the cost of both the operation itself and any wasted prefetching. Profiling concentrates on random accesses as multi-tenant key-value caches are mostly random: first, because multiple tenants ac-

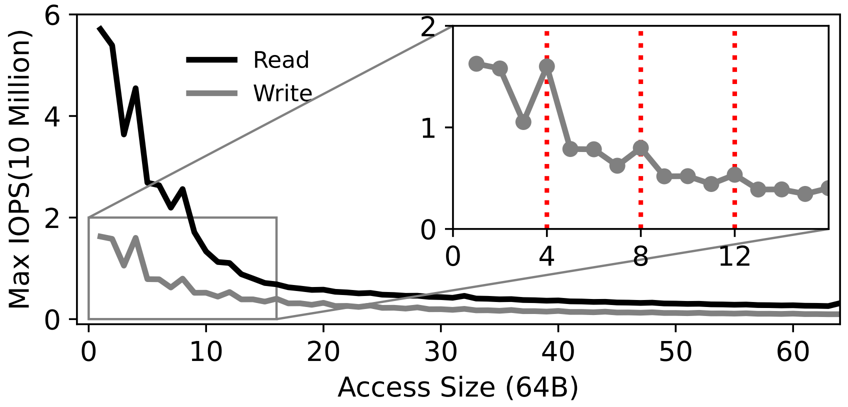


Figure 5.3: **MaxIOPS Profile.** *This figure shows example profile of maximum IOPS for random reads and writes of different sizes on our 2× Intel Optane DC PM system.*

cess PM simultaneously (in different address spaces), their requests are interleaved; second, keys tend to be mapped to arbitrary PM locations based on their time-to-live and size [52, 240]. The profiler stops at request sizes of 4KB which obtain the device's maximum bandwidth.

Figure 5.3 shows the profiled MaxIOPS for reads and writes as a function of request size. As shown, writes have lower IOPS and thus a higher cost per operation than reads. While larger requests generally have lower IOPS, there is a complex relationship with the minimal PM access size: for example, a 64B random store has a similar maximum IOPS as 256B, the minimum PM access size; accesses that are not aligned to 256B have lower MaxIOPS.

At runtime, Nyx tracks the PM usage of each cache instance. When a cache instance accesses PM, Nyx looks up the MaxIOPS for this operation and size, and increments a cost counter for this cache instance by $\frac{1}{\mathrm{MaxIOPS}}$. To reduce synchronization overhead, these counters are maintained per-thread and only lazily combined when needed (e.g., for responding to a resource usage query from Nyx Controller).

While the CPU cache can theoretically introduce errors in PM cost estimation, these errors are negligible for Nyx. First, since CPU prefetching waste depends in part on spatial locality, the profiler mimics the random accesses of cache instances that have little sequentiality. Second, given a cache instance that uses NT-store (as in Nyx-Pelikan), the CPU cache has no effect on stores. Finally, although a PM load could be served in the CPU cache and never access PM, in multi-tenant caches few PM loads hit in the CPU cache: because each instance's working set is typically tens of GBs [95, 239] (and there are many instances), there is little temporal locality in CPU caches of tens of MBs. More intricate cost models for cache instances with spatial (e.g., scan) and temporal locality (e.g., bursty retries) are left for future work.

Interference Analysis: When multiple cache instances are co-located, Nyx determines which instance most impacts another. For example, when an efficient QoS implementation observes that an affected client W is not meeting its guarantee, it will iteratively slow down the one competing client that will produce the greatest benefit for W. In PM-based systems, unlike DRAM, these interactions are difficult to identify because an affected client may be impacted more by a low-bandwidth client than a high-bandwidth client. The amount of interference is due to complex scheduling within the PM device; as future generations of PM devices become available, which clients interfere with which others may change. Therefore, Nyx assumes no prior knowledge of these interactions.

Nyx determines which client is interfering the most with the affected client with a runtime micro-experiment. Given affected client W and several competing clients, Nyx iteratively throttles each competing client by X for some metric of interest while measuring the impact on client W. The throttled client that helps W attain the greatest performance improvement is identified as the client that interferes with W the most. The value of X is configurable, as is the metric (e.g., throughput, average latency, or tail latency). Nyx uses simple pruning techniques to throttle only the clients with the highest resource usage. Optimizations for reducing micro-experiment times (e.g., focus on different client subsets in different trials) are left for future work.

SlowDown Estimation: Nyx determines the slowdown that each client experiences at runtime by calculating $\frac{T_{alone}}{T_{share}}$; T_{alone} is the client's performance (for some metric of interest) when it is running alone, and T_{share} is its current performance in the shared environment. As we assume no special hardware, Nyx uses an approach similar to previous work [157].

First, to learn T_{alone}, Nyx briefly pauses all other clients; T_{alone} is updated on a regular basis (e.g., 1s) or whenever a workload change is observed. Second, slowdown is periodically calculated using a runtime measurement of T_{share}. As we will show, at the cost of a small loss of bandwidth and increase in tail latency, this solution adequately approximates slowdown without hardware support. The impact of the pause can be reduced for workloads that do not change frequently.

5.2.4 Nyx Sharing Policies

Nyx implements four popular sharing policies. We describe how these policies leverage the mechanisms of Nyx for PM.

Resource Limit: Nyx can limit the amount of the PM resource used by each client in multi-tenant caching, isolating the performance of clients from one another. Our policy defines resource limits in terms of standard operations, similar to Google Cloud's memcache [27] (e.g., 1000 1KB random reads per second, or 1MB/s random reads).

As shown in Algorithm 2, Nyx provides resource limits for each client epoch by epoch, extending existing approaches [243]. Each epoch, Nyx monitors the resource utilization of each client; if a client reaches its limit for this epoch, its accesses to PM are delayed until the next epoch. When the epoch ends, the throttling value for each client is reset to zero. The

Algorithm 2: Resource Limit The gray area denotes unique functionality used to deal with PM issues

```
   EpochLen: ticks in an epoch (e.g. 100), TickLen: (e.g. 10ms)
   A.getResCounter(): query A's Nyx-Lib for resource usage
   A.setThrottling(t): add t×10ns delay to each access of A
   ResAssigned[1..N]: each cache's assigned resource per epoch
1  while true do
       # Step 1: Begin an epoch and set all cache throttling to 0
2      foreach cache A do
3          A.setThrottling(0)
           InitResCounter[A] = A.getResCounter()
       # Step 2: Monitor resource utilization and pause clients who have used up
         their allotted resources.
4      while Epoch is not completed do
5          SleepFor(TickLen)
6          foreach cache A do
7              ResUsed = A.getResCounter() - InitResCounter[A]
8              if ResUsed > ResAssigned[A] then
9                  A.setThrottling(INFINITE) # Pause
```

implementation allows the administrator to configure the *epoch* and *tick* length to trade-off the overhead of checking counters with reaction time.

Quality-of-Service: Nyx can ensure that latency-critical (LC) tenants meet a service-level-objective while maintaining high PM utilization for best-effort (BE) tenants on the same server. As in earlier work [120, 121], admission control prevents workloads with unachievable QoS targets and space-allocation provides the necessary hit ratio.

As shown in Algorithm 3, Nyx employs an approach similar to Parties [111] and Caladan [135]: for each LC client, the difference between the guaranteed and the current performance is tracked; when the guarantee is violated (i.e., negative slack), a competing tenant is throttled.

Nyx differs in how it identifies the client to be throttled. Caladan always throttles the BE tenant with the maximum bandwidth (LLC

Algorithm 3: QoS The gray area denotes functionality for PM. We omit code to rollback throttling when the action violates any LC task's target.

```
ExperimentStep: a cache's throughput expense pays for an interference
  analysis experiment. (e.g. 500MB/s)
1  while true do
       # Step 1: Monitor each client's SLO slack
2      foreach cache A do
3          slack[A] = (A.target - A.latency) / A.target
4      S = cache with the smallest slack
       # Step 2: Protect clients violating SLO
5      if slack[S] < 0 then
6          if S is throttled then
7              throttle down S
8          else
               # Step 2.1: Pick candidates to throttle
9              if there are BE caches then
10                 candidates = top 3 resource usage BE
11             else
12                 candidates = top 3 res usage LC, slack > 0.2
13                 if all LCs have little slack then
14                     candidates = LC with the most slack
               # Step 2.2: Find the most interfering client
15             I = getLargestInterference(S, candidates)
16             throttle up I
17     else if slack[S] > 0.2 then
           # All caches have slack -> relax throttling
18         throttle down every cache
19 Function getLargestInterference(S, Candidates):
       # Find the tenant who will most improve S at the same expense
         (throughput)
20     If there is only one client in Candidates, return the client
21     foreach C in Candidates do
22         throttle up C by ExperimentStep
23         track S latency change after the experiment
24         restore all throttle to previous state
25     return L who helps S get the largest improvement
```

Algorithm 4: Fair Slow Down

```
A.getSlowDown(): return A's current performance / $T_{alone}$
while true do
    if $T_{alone}$ info is older than P sec then
        foreach cache A do
            refreshTalone(A)
    # Adjust throttling to equalize slowdowns
    foreach cache A do
        SlowDown[A] = A.getSlowDown()
    find cache L and S with the largest and smallest slowdowns
    unfairness = SlowDown[L] / SlowDown[S]
    if unfairness > UnfairnessThreshold then
        throttle down L and throttle up S
        FairIntervals = 0
    else
        # With fair slowdown, try to improve utilization
        FairIntervals ++
        if FairIntervals > FairIntervalThreshold then
            throttle down all caches
Function refreshTalone(A):
    A.setThrottling(0), and pause every other cache
    A.$T_{alone}$ = measure A throughput
    restore throttle of all caches to previous state
```

misses), whereas Nyx throttles the BE or LC cache that most improves the LC cache, for the same expense across competing tenants. The implementation allows the administrator to configure *ExperimentStep*, allowing a balance between aggressive throttling and faster convergence.

Fair Slow Down: Nyx can achieve fairness in terms of equalized slowdown across caches. As in Algorithm 4 [127, 215], Nyx minimizes (MaxSlowDown/MinSlowdown) by gradually increasing the throttling of the MinSlowDown cache and decreasing the throttling of the MaxSlowDown cache. The tuning process is terminated when the unfairness metric falls under an UnfairnessThreshold. The implementation periodically (every P seconds) refreshes the estimate of the stand-alone performance (T_{alone}) for each client. Administrators can customize P to balance between lower overhead and faster adjustments for dynamic workloads.

The policy can be generalized to guarantee weighted slowdowns and a hard limit on some cache's slowdown. For the hard limit, Nyx tracks the particular slowdown at runtime and throttles other caches when the hard limit is exceeded.

Proportional Resource Allocation: Nyx implements proportional sharing with actual proportional resource allocation (instead of simple bandwidth allocation) and with interference-aware idle resource redistribution. Nyx ensures that each cache achieves performance equal to or better than accessing PM alone for a given amount of time (time-sharing [220]). For example, if a cache has a weight of 2 out of 3, then it is guaranteed to obtain at least 2/3 of its stand-alone performance.

Nyx first allocates resources (not bandwidth) proportionally to each cache and enforces the resource limit during an epoch (Algorithm 5). We assume cache space has been allocated proportionately. Following an epoch, Nyx forecasts each tenant's desired amount of resources: a tenant that did not use all its given resource may donate idle resources, whereas a tenant that used all assigned resources may consume more (a simple linear model predicts desired resources [243]).

Nyx provides interference-aware resource donation (Option 2 in the Alg.). On PM, idle resource redistribution faces the difficulty that the donated resource may severely interfere with the original donor's performance. For example, as shown in Section 5.3.5, if a get-heavy cache A donates idle resources to a write-heavy cache, the new write traffic can dramatically harm A's performance. To prevent this interference, Nyx re-allocates resources in increments, stopping when the donating cache's slowdown is near its lower bound; if the slowdown exceeds the lower

Algorithm 5: Proportional Resource Allocation The slowdown refreshing code is omitted.

```
   DonateStep: step to donate idle resources (e.g. 10%)
   TotalResource = 1
1  while true do
       # Step 1: Enforce and track resource usage in an epoch
2      Begin a New Epoch
3      foreach cache A do
4          Enforce A uses resource <= ResourceAssigned[A]
5          if A depleted resources, record how long: TimeUseUp[A] (e.g.
             half of the epoch)
6          if A left idle resources, record ResourceUsed[A]
7      End of the Epoch
       # Step 2: Redistribute Idle resources
8      foreach cache A do
9          if A has idle resources then
               # Option 1: Donate all extra resources
10             DesiredResource[A] = ResourceUsed[A]
               # Option 2: Interference-aware resource donation
11             if A.getSlowdown() < TotalWeight / A.weight then
                   # Donate a step when within slowdown limit
12                 DesiredResource[A] = Max(ResourceAssigned[A] * (1 -
                   DonateStep), ResourceUsed[A])
13             else
                   # Revoke a step when under slowdown limit
14                 DesiredResource[A] = Min(ResourceAssigned[A] * (1
                     + DonateStep), TotalResource * A.weight/ TotalWeight)
15         if A depleted resources: DesiredResource[A] =
           ResourceAssigned[A] / TimeUseUp[A]
16     ResourceAssigned[1..N] = Allocate resources proportionally based
         on weight and desired resource
```

bound, a portion of the donated resources are returned. Thus, Nyx guarantees the "time-sharing" lower bound while maximizing resource utilization. The implementation allows the administrator to set *DonateStep*, balancing quick idle resource donation and the proportional guarantee.

With Admission Control and Capacity Allocation: In a nutshell, cache instances are 1) admitted, 2) allocated space, and 3) governed by Nyx. A PM free-space check, for example, suffices for resource limiting as admission control for a cache; QoS policy requires logic like [120, 121] to predict SLA compliance given existing caches. The cache size is then determined. For instance, it can be set based on the instance's price tier; to enforce QoS, administrators can profile a client's hit-rate v.s. cache space relationship [214] and allocate enough space to meet SLAs. While running, Nyx assumes the admission logic is correct and is unconcerned about the space allocated.

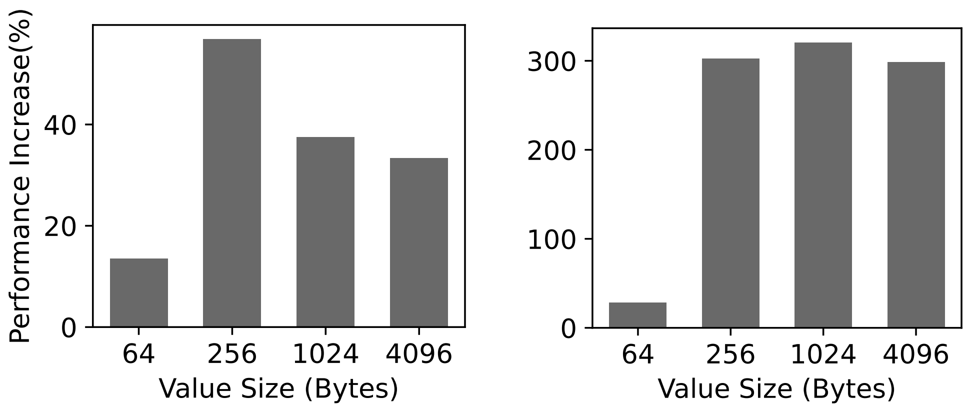
5.2.5 Cache Instances: PM-Optimized Pelikan

Nyx has been designed to handle any in-memory key-value store; our current implementation is built upon Pelikan – Twitter's in-memory KV cache [61, 240]. We describe the original Pelikan and optimizations for higher PM performance.

Pelikan (SegCache [240]) maintains a hash table for indexing and segments for storing key-value pairs. Each segment includes items, where each item is a tuple of (key, value, metadata). On a get operation, Pelikan hashes the key to find items. Because of conflicts, multiple keys are likely to be read for a single get. Thus, Pelikan must compare each read item with the key; if the keys match, the value is returned.

When the default version of Pelikan is configured for PM, the hash index is kept in DRAM and the segments in PM. However, this placement is inefficient due to the frequent key accesses in PM: the keys in caching workloads are often much smaller [239] than the granularity of PM access (256B), and small reads perform relatively poorly on PM [238].

Nyx-Pelikan addresses this by separating keys (and metadata) from values into different segments; the keys (and metadata) are placed in DRAM and the values in PM. This design requires DRAM for keys and



(a) Get (b) Write

Figure 5.4: **Optimization: Nyx-Pelikan.** *This figure shows Nyx-Pelikan performance. (a) presents Nyx-Pelikan Get (single-thread) throughput improvement due to key-value separation. (b) presents Nyx-Pelikan Write (replace, 8 threads) improvement due to changing stores to NT-stores.*

metadata, which works well because they are typically much smaller than values [238].

As shown previously in Table 5.2, because non-temporal stores to PM can provide much greater throughput than conventional stores, Nyx-Pelikan uses NT-store. Although non-temporal stores may not benefit from temporal locality in the CPU cache, this loss is negligible on large-scale caching workloads which typically have large working sets. As shown in Figure 5.4, Nyx-Pelikan improves Pelikan Get performance by up to 55% and set performance by up to 3×.

5.2.6 Nyx Parameter Values

The values of Nyx's parameters affect its behavior; as previously stated, the appropriate settings depend on the tradeoffs made by administrators. Nyx enables users to configure all of these parameters while also setting defaults.

Nyx follows existing guidelines [127, 215, 243] for policy parameter values' selection. For resource limiting, Nyx uses 10 ms tick and 100 ticks per epoch to limit resource usage offset to 1%. For fair slowdown, Nyx sets the T_{alone} refresh interval to one second to achieve a relatively quick response to workload changes and a within 2% overhead (§5.3.1).

Nyx provides defaults for newly introduced parameters via sensitivity tests (§5.3.7). Nyx QoS uses 500MB/s ExperimentStep because it is the smallest step that produces good interference analysis. In interference-aware resource donation, Nyx sets a 10% DonateStep to balance quick donation and steady donator performance. Nyx sets 10ns throttling delay granularity for fine-grained access rate regulation, which is an order of magnitude less than 100ns PM latency. We will discuss potential optimizations like dynamic/adaptive parameters and automatic parameter value selection in §5.4.

5.3 Evaluation

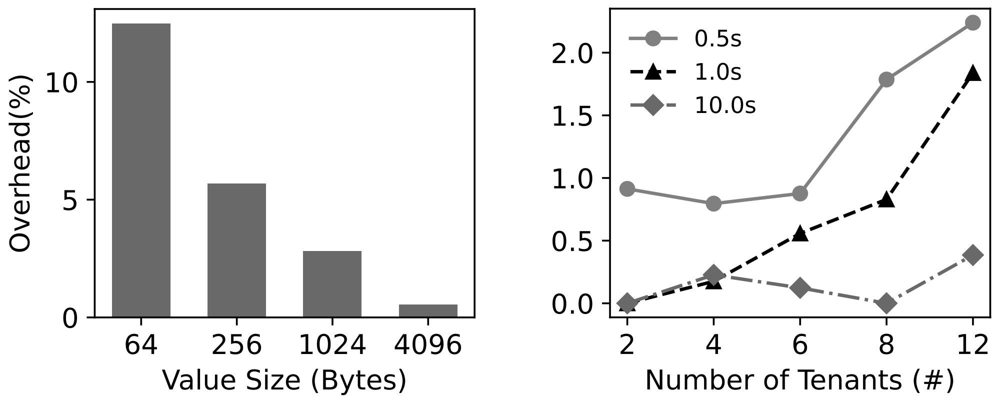
We evaluate the overhead of Nyx's mechanisms and how well Nyx provides the sharing policies of resource limit, QoS, fair slowdown, and proportional resource allocation.

Setup: We use a 16-core, single-socket Intel Xeon Gold 5128 CPU @ 2.3GHz server (Ubuntu 18.04), with a 22 MB L3 Cache, 2x16GB DRAM, and 2x128GB Intel Optane DC PM in app direct mode. We mount an ext4 file system in DAX mode on the PM.

Synthetic Workloads: We begin with synthetic workloads to illustrate key features. Unless specified, the workloads have uniform random accesses to each cache instance, a working set of 10GB per instance, and 4B keys and variable-sized value. To focus on PM accesses, we use get workloads with a high hit ratio (>99 percent). We use in-place replacement for write-heavy workloads; a cache write implies a replace. The cache is

Trace	Type	Avg.Key/Value Sizes(B)	Operations (Get/Write ratio)
S1	Storage	36/799	0.86/0.13
C1	Computation	67/2439	0.93/0.07
C2	Computation	18/67485	0.52/0.48

Table 5.3: **Twitter Traces.**



(a) Regulation, Accounting Overhead (b) Slowdown Estimation Overhead

Figure 5.5: **Mechanisms Overhead.** *This figure shows the overhead of Nyx mechanisms. (a) shows Nyx request regulation and resource usage accounting overhead (throughput). It is measured with 8-threads get-only caches. A similar percentage of latency overhead was observed. (b) shows Nyx slowdown estimation overhead (throughput). It is measured with 1ms* T_{alone} *pausing time, different number of clients (x axis) and different frequency (0.5/1/10s) of updating* T_{alone} *for all caches.*

warmed to begin.

Realistic Workloads: We conclude with three large-scale cache traces from Twitter [239] (Table 5.3). The traces cover caches with various value sizes (799B to 67845B) and get-percentages (93% to 52%). We pre-load one million operations from the traces and loop through them.

5.3.1 Mechanisms Overhead

Request Regulation and Resource Usage Estimation: With Nyx, each PM access incurs a call into Nyx-lib, throttling logic, and resource accounting. Figure 5.5.a shows this can add up to 12% overhead for extremely small value sizes (e.g., a cache line), but less than 6% for access sizes above 256B. Given the benefit of request regulation and resource usage accounting, we believe this overhead is justified.

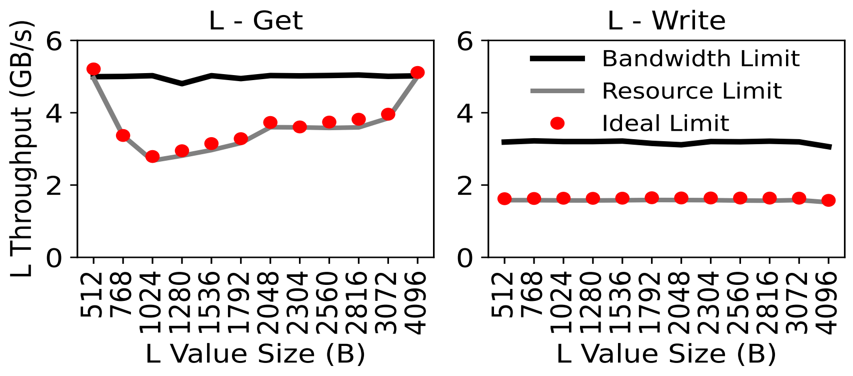
Interference Analysis: Determining the most interfering client takes longer than simply selecting the client with the greatest bandwidth due to the lag necessary to observe tail latency. In Section 5.3.3 we will demonstrate the benefit of trading increased analysis time for more precise information.

SlowDown Estimation: The overhead of slowdown estimation is influenced by the time to measure T_{alone} per instance, the frequency of this measurement, and the number of cache instances. We determined that 1ms is a sufficient pause time to accurately determine T_{alone} for a client. Figure 5.5.b shows that calculating T_{alone} for up to 12 instances adds less than 2.5% overhead, even when performed every 500ms.

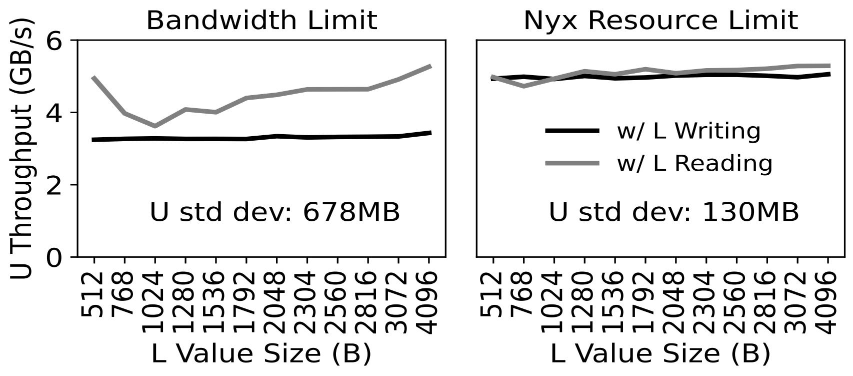
5.3.2 Resource Limiting

We demonstrate that Nyx can enforce a true resource limit on PM, in contrast to an approach based only on bandwidth. We begin with a workload containing one unlimited (U) cache and one limited (L) cache. Cache U is a get-heavy cache instance, while Cache L changes: get-only or write-only, with varied value sizes. L has a resource limit of 1.25M 4KB random load OPS, or 42% of the total device resource given that MaxIOPS for 4KB random loads is 3 Million. Defined in terms of bandwidth, this equates to 5GB/s for these 4KB random loads; however, this IOPS limit results in different bandwidths for other workloads.

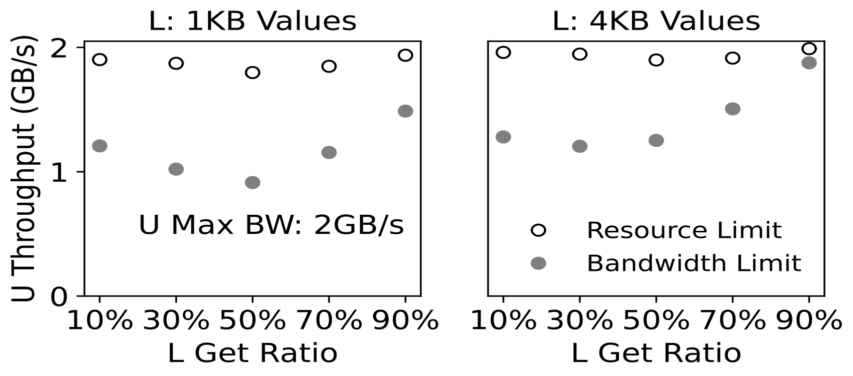
Figure 5.6.a shows the bandwidth of L; the target IOPS, in which no more than 42% of the device resource is used, is shown in red. As desired, Nyx always limits L's throughput to the target limit, regardless of L's access pattern (determined by value sizes and read/write). In contrast,



(a) L Throughput



(b) U Throughput



(c) U Throughput + Broader L Setups

Figure 5.6: **Resource Limit: Cache U (unlimited) + Cache L (limited).** *This figure compares the Nyx Resource Limit policy behaviors with the classic Bandwidth Limit policy behaviors. Cache U is get-only. In (a), we show Cache L throughput when its resource limit is 5GB/s (1.25M 4KB random load OPS, or 42% of the total device resources). The red dotted line represents L's performance under the "ideal limit", which is calculated as 42% of the current access pattern's MaxIOPS. L is get-only or write-only, and its value sizes varies (x axis). In (b), we show cache U's performance when colocated with the same L in (a), comparing bandwidth limit and Nyx resource limit. In (c), we investigate additional L setups: 1KB/4KB value sizes and 10% - 90% gets. U is a lighter cache than (a) and (b). The label indicates U's max bandwidth when colocated with a 5GB/s cache instance (4KB-value, get-only).*

a policy based only on bandwidth mistakenly allows L to significantly exceed the target limit, up through the maximum bandwidth of 5GB/s. When L is get-only, this problem is most noticeable when the value size is around 1KB; as previously noted, 1KB accesses result in significant CPU prefetching waste not captured by software-level bandwidth accounting. On the other hand, Nyx's MaxIOPS cost model accurately captures resource usage. Similarly, bandwidth cannot capture PM write cost and fails to properly limit L's throughput.

The impact on the unlimited client (U) is shown in Figure 5.6.b for the same L workloads. With the bandwidth policy, U's performance depends on L's access pattern. Due to asymmetric read/write cost of PM, whether L performs reads or writes significantly impacts U; similarly, the varied prefetching waste of each access pattern causes up to 45% impact on U. In contrast, Nyx provides U with steady and predictable performance, regardless of L's access pattern: across all of L's workloads, the standard deviation of U's performance is only 130MB/s (bandwidth limit's deviation is 678MB/s). Finally, Figure 5.6.c shows that when the percentage of gets in L is varied, Nyx provides steady performance for U, whereas a PM-oblivious bandwidth-based approach does not.

Figure 5.7 demonstrates Nyx's resource limiting behaviors as the cache hit rate varies. As shown, Nyx restricts PM resource usage from (get) hits. Misses in look-aside caches (e.g., Pelikan) are simply returned after checking the index (in DRAM) and do not use PM resources, so they are not limited.

5.3.3 QoS-Aware

Nyx can provide QoS guarantees for latency-critical (LC) caches while providing high utilization to best-effort (BE) caches with interference-aware regulation; in contrast, a PM-oblivious approach such as that in Caladan may not be able to deliver the same performance to the BE cache.

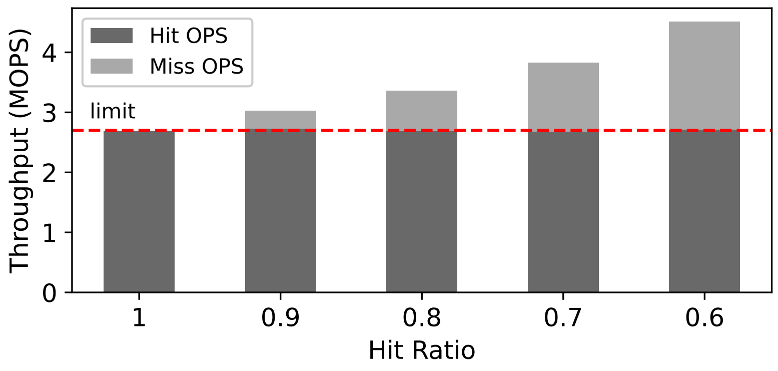
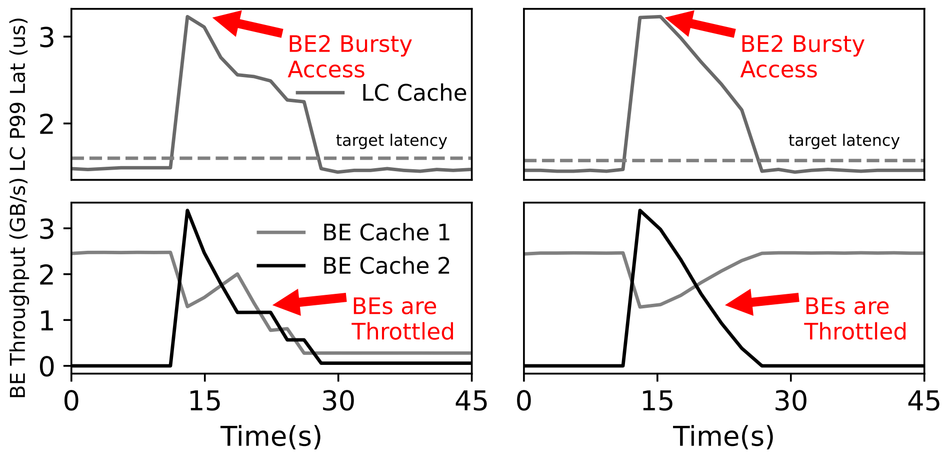


Figure 5.7: **Resource Limit: Behaviors with a Varying Hit Rate.** *This figure shows Operations Per Second (OPS) for a Cache with 1KB Get-only workloads when resource limit is 5GB/s. We vary the workloads with different working sets to achieve a different hit rate; note there is no insertion after each miss.*

For comparison, we implemented the Caladan approach in Nyx-Caladan.

Figure 5.8 shows an LC cache (P99 latency target of 1.5µs) colocated with two BE caches: BE1 is get-heavy, BE2 is write-heavy. Initially, when BE2 has low throughput and BE1 has moderate throughput of 2.4GB/s, LC meets its P99 objective; however, at 12s, BE2 performs many bursty writes, causing LC's P99 latency to exceed 3µs and violate its target. Both Nyx-Caladan and Nyx resolve the situation by iteratively throttling a BE cache. Nyx-Caladan throttles the cache currently consuming the most bandwidth, shown in the left two subfigures; as a result, Nyx-Caladan throttles both BE1 and BE2, resulting in $\times 6$ less bandwidth for BE1. Nyx, on the other hand, identifies the cache that most interferes with LC as BE2, the write-heavy cache. As a result, Nyx stabilizes to throttling only the correct interference source; after 28 seconds, only BE2 is throttled, and BE1 returns to its original throughput. To summarize, Nyx provides high utilization for multiple caches while guaranteeing each target.

Nyx's convergence time of tens of seconds is similar to prior work such as Parties [111]: the majority of the converging time is spent monitoring

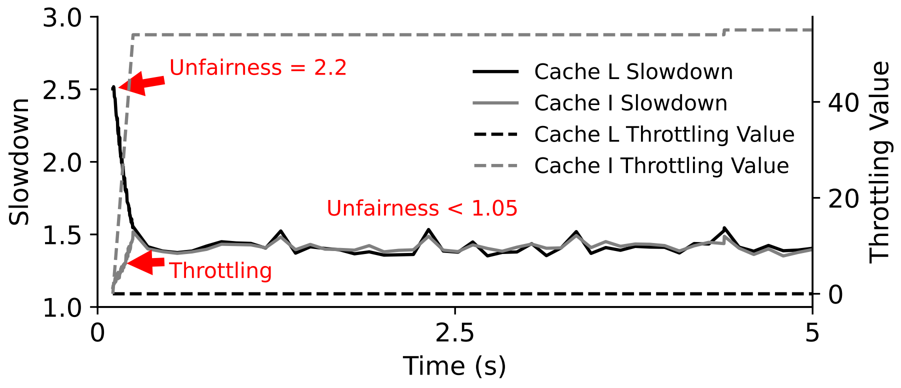


(a) Nyx-Caladan (b) Nyx

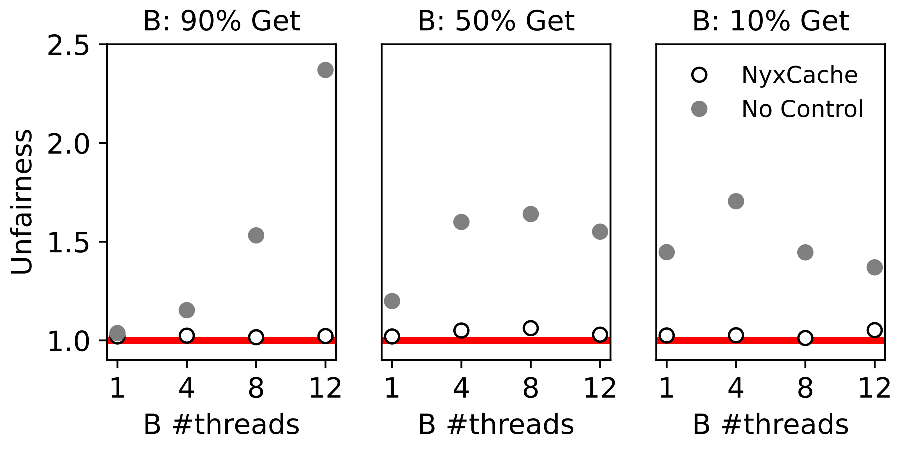
Figure 5.8: **QoS: Nyx-Caladan vs. Nyx Tuning.** *This figure shows how Nyx (b) and Nyx-Caladan (a) throttle BE caches to ensure LC cache P99 latency. We plot LC P99 latency (top figures) and BE caches throughput (bottom figures) over time (x axis). LC cache is colocated with two BE caches; BE1 is get-heavy, B2 is write-heavy (i.e., more interference to LC). BE2 has burst at 12s, breaking LC latency targets. Nyx-Caladan (a) throttles the highest-bandwidth client, whereas Nyx (b) throttles the client with the most interferences to LC. Nyx-Caladan incorrectly throttles BE1, resulting in $\times 6$ less bandwidth for BE1.*

tail latencies. As in Parties, Nyx measures tail latency for 500ms because shorter intervals can result in noisy measurements. We leave faster tail latency measurement at network packet queues (as utilized in the original Caladan [135]) for future investigation.

Our experiments reveal that Nyx has an intriguing effect on convergence time: as shown in the Figure, Nyx can bring the LC cache to its target performance in a comparable amount of time to just selecting the cache with the highest bandwidth (which does not require any micro-experiment time). The implication of these results is that, rather than simply acting quickly and throttling any competing instance, Nyx acts correctly and throttles the source of the interference.



(a) Tuning Process (Light + Intensive)



(b) L + various B

Figure 5.9: **Fair Slowdown.** *This figure shows behaviors of the Nyx Fair Slowdown policy. (a) shows how Nyx equalizes slowdown over time for two cache instances (a light one (L) and an intensive one (I)). Both cache instances are get-heavy. (b) shows the unfairness metric when colocating L (a light get-heavy cache) with different B instances (get-heavy -> write-heavy, and light -> intensive). Unfairness = MaxSlowDown / MinSlowDown, the more close to 1, the more fair.*

5.3.4 Fair Slowdown

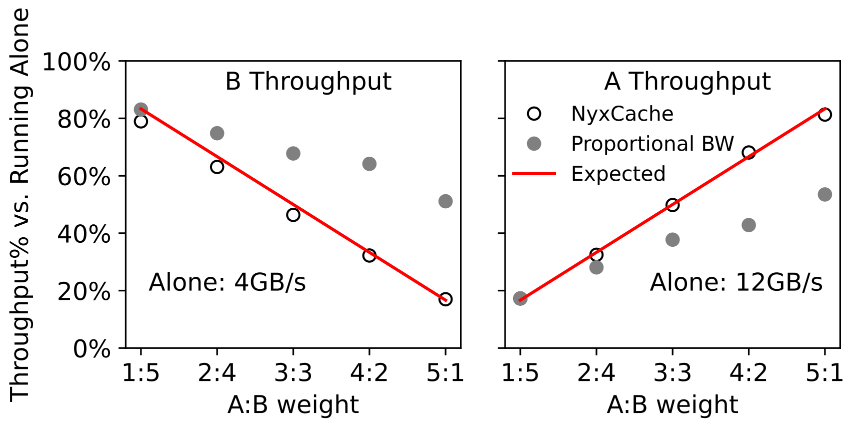
Nyx implements fair slowdown by iteratively regulating requests according to the measured slowdown of each client (i.e., $\frac{T_{alone}}{T_{share}}$). Figure 5.9.a shows Nyx's tuning given colocated light and intensive get-heavy caches. Initially, the slowdown of the light cache is 2.2 times higher than that of the intensive cache. Over time, Nyx dynamically increases the throttling of the cache with the minimum slowdown and decreases throttling for the cache with maximum slowdown. Relatively quickly, both caches converge to a slowdown near 1.5 and the unfairness metric of $\frac{MaxSlowdown}{MinSlowdown}$ settles near 1.05.

Figure 5.9.b shows Nyx's fair slowdown policy on a range of caches. Cache L remains a light get-heavy cache; Cache B varies the number of threads and can be get-heavy, 50% mixed, or write-heavy. Without Nyx, L can experience dramatically unfair slowdown (due to PM's complex performance); for example, colocating A with a multi-threaded get-heavy cache B gives unfairness near 2.4. In contrast, Nyx achieves fair slowdown ($<$ 1.05 unfairness) for all 12 cases.

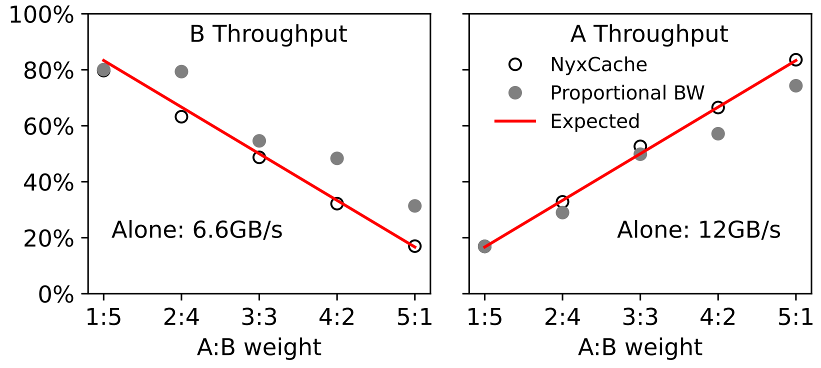
5.3.5 Proportional Resource Allocation

Nyx achieves proportional resource allocation and guarantees a time-sharing lower bound while performing idle resource re-distribution. We begin with simple scenarios in which two caches that use all their assigned resources are colocated. The scenarios in Figure 5.10 vary the desired proportional share for A and B along the x-axis; the red line indicates the ideal proportional throughput given their throughput when run alone. Figure 5.10.a shows that a PM-oblivious bandwidth approach cannot guarantee a proportional share; in particular, the write-intensive B cache obtains up to 3$\times$ more throughput than desired and the get-intensive cache A suffers significantly (~40%). However, by correctly estimating resource usage, Nyx delivers the desired allocation to each cache. Figure 5.10.b shows a similar effect occurs when efficient-get (value: 4KB) and inefficient-get (value: 1KB) caches are colocated.

Proportional allocation is more challenging when there are idle re-



(a) A: Get-heavy, B: Write-heavy



(b) A: Efficient-Get, B: Inefficient-Get

Figure 5.10: **Proportional Sharing.** *This figure shows Nyx Proportional Sharing policy behaviors. We colocate two caches A and B. (a) shows A (get-heavy cache) and B (write-heavy cache)'s throughput with different weight configuration. The labels indicate running alone throughput of A and B. With bandwidth allocation, B surpasses its allotted proportional performance. (b) shows A (efficient get-intensive cache, 4KB value sizes) and B (inefficient get-intensive cache, 1KB value sizes).*

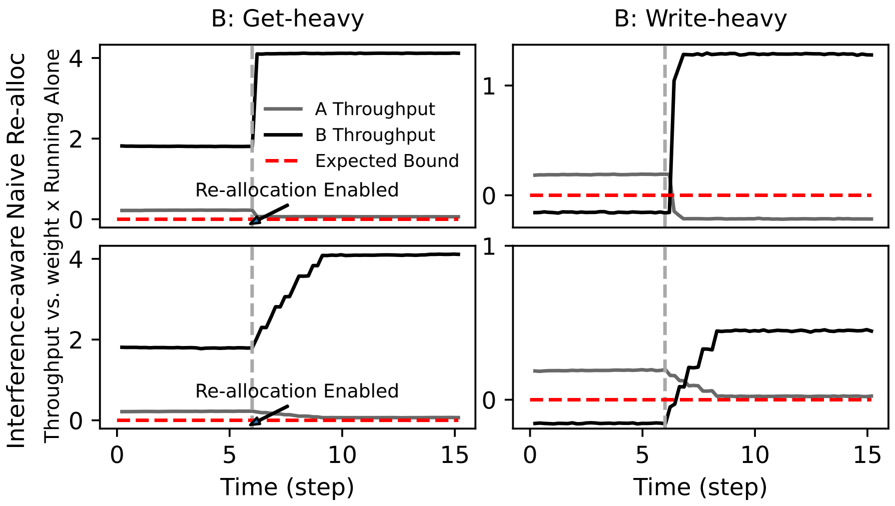


Figure 5.11: **Extra Resource Re-distribution in Proportional Share: Problem of Naive "Donate All" Strategy.** *This figure compares naive "Donate All" strategy (top figures) and Nyx interference-aware donation strategy (bottom figures). We colocate two caches A, B and weight A:B as 2:1. This figure shows cache A (light, get-heavy) throughput before and after it donates its extra allocated resource at time step 6. A has a 75 percent idle resource. Y axis represents the normalized difference between A throughput and A's running alone throughput. When cache B is get-heavy (the top-left figure), A gets nominal performance drop due to donation. However, when cache B is write-heavy (top-right figure), donating cause severe slowdown for A. Unlike naive extra re-distribution, Nyx (two bottom figures) ensures that tenant A's performance is always more than two-thirds of its running alone performance. TimeStep = 2ms.*

sources to be redistributed. Figure 5.11 shows two caches A and B, where A uses only 25% of its share. When B is get-heavy (left-top subfigure), A can donate all its idle resources to B; A's performance is slightly degraded, but B receives substantially higher throughput. However, when B is write-heavy (right-top subfigure), if A donates all its idle resources, the higher throughput of B substantially interfere with A, breaching A's time-sharing lower bound (2/3 of A's stand-alone throughput). Therefore, Nyx does not perform naive donation; instead, Nyx donates idle resources in increments while monitoring each cache's slowdown. As

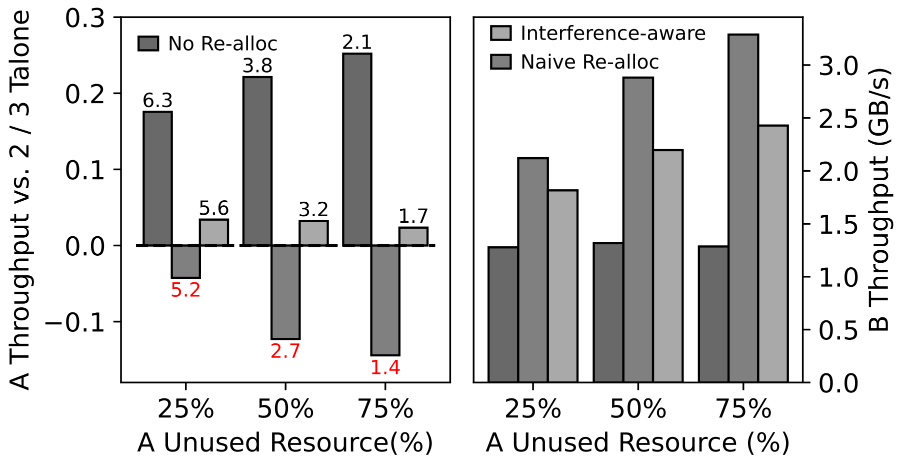


Figure 5.12: **Extra Resource Re-distribution in Proportional Share: Re-distribution between Get and Write-heavy Caches.** *This figure compares behaviors of i) no extra resource re-distribution (No Re-alloc), ii) Nyx interference-aware extra resource re-distribution (interference-aware), and iii) naive "Donate All" re-distribution (naive re-alloc). We set cache weight A:B as 2:1. A is get-heavy and B is write-heavy. This figure shows A's slowdown (left figure) and B's throughput (right figure) before and after A donating extra resource.The label indicates absolute throughput number. Naive extra resource allocation can easily break isolation guarantee, while Nyx always ensures it.*

shown in the bottom two graphs, Nyx guarantees the time-sharing lower bound for each cache while improving utilization.

We next examine workloads varying the percentage of idle resources in Cache A. When cache B is get-heavy, all of A's idle resources can be safely redistributed to B, and Nyx achieves the same performance for cache B as simple donation (Figure not shown due to space limit). However, when cache B is write-heavy, simple donation of A's idle resources to B violates A's time-sharing bound (Figure 5.12); Nyx accurately protects cache A's performance while still improving the performance of cache B relative to no donation.

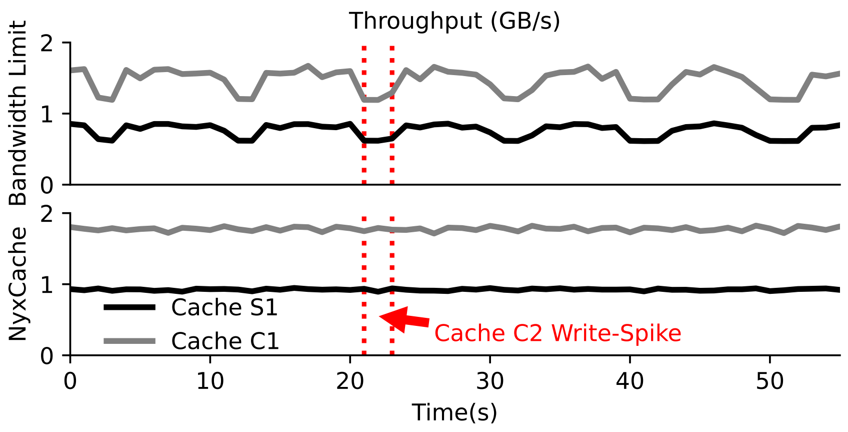


Figure 5.13: **Realistic Traces: Protecting Caches from Write Spikes.** *This figure shows the performance of Cache S1 and C1 when colocated with Cache C2. Cache C2 has write spikes. Nyx (bottom figure) can isolate write spikes, whereas bandwidth limit approach cannot (top figure).*

5.3.6 Realistic Traces

Nyx provides isolation for realistic workloads. We demonstrate use cases for resource limiting and slowdown limiting.

In production workloads, write spikes are common; for example, when a cache is used for ML models, write spikes occur with model parameters are regularly refreshed [239]. Figure 5.13 shows how Nyx can isolate caches S1 and C1 from (added) write spikes in cache C2. If resource limiting is based only on the bandwidth of C2, S1 and C1 suffer when C2 experiences write spikes. However, Nyx's resource-limit policy can cap C2's resource usage (at 4GB/s, defined as 1M 4KB random load OPS) to keep S1 and C1 steady.

Nyx can also protect the performance of critical caches. To encourage tenants to use multi-tenant PM environments, some caches must be guaranteed performance similar to exclusive use of the PM device. In the experiment shown in Figure 5.14, S1 (the critical cache) is colocated with C1 and C2 which have diurnal patterns [239]. With no control (gray

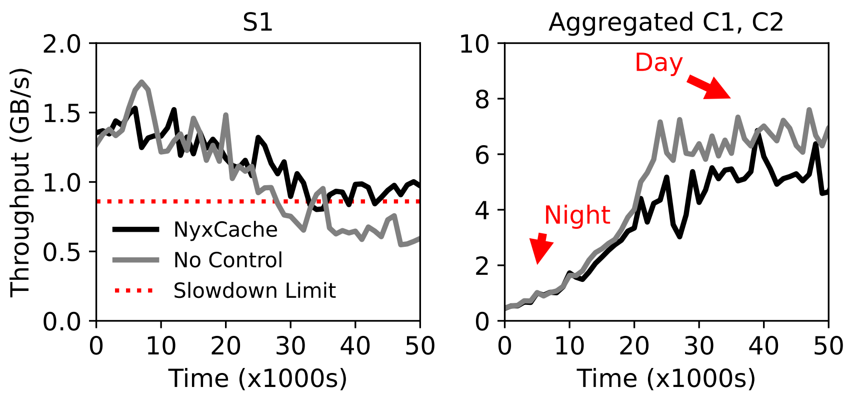


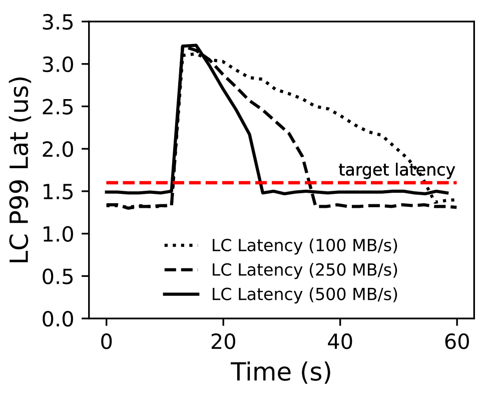
Figure 5.14: **Realistic Traces: Limiting S1 Slowdown During Day and Night.** *This figure shows the performance of Cache S1 over time (left figure, we guarantee S1's slowdown is always smaller than 1.5×). S1 is colocated with C1 and C2; both C1 and C2 have a strong diurnal pattern (light during the night, and intensive during the day as shown in the right figure). Without Nyx, S1 performance plummets during the day (because the impact from C1 and C2), discouraging sharing. However, Nyx can always offer reasonable performance (e.g. within 1.5× slowdown vs. running alone). The red line represents S1's performance guarantee.*

lines), the performance of S1 drops below its target during the day due to the heavy accesses of C1 and C2. However, Nyx can establish a hard limit of slowdown (e.g., 1.5) for S1. As observed, Nyx keeps S1 performance loss within a fair range.

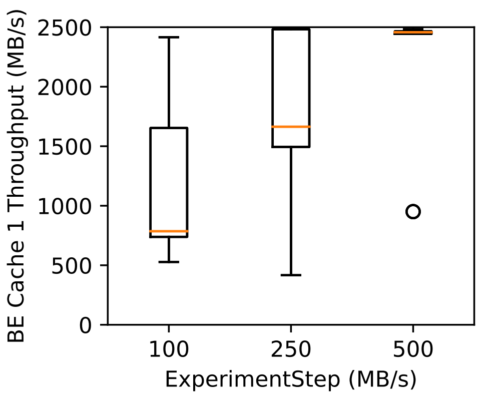
5.3.7 Parameters Sensitivity Analysis

Here, we present the sensitivity analysis of Nyx behaviors with different ExperimentStep and DonateStep values.

The ExperimentStep affects the Nyx interference analysis's accuracy. As shown in Figure 5.15.b, using the same configuration as Figure 5.8, a smaller ExperimentStep is more likely to result in a lower BE 1 final throughput. When ExperimentStep is small, the tail latency change is more likely to be due to measurement noise rather than interference, leading to a less accurate interference analysis. Our experiments suggest an



(a) Convergence Time



(b) BE 1 Final Throughput

Figure 5.15: **QoS: ExperimentStep Sensitivity Analysis.** *Same as in Figure 5.8. (a) shows how fast Nyx QoS can ensure LC P99 latency varying ExperimentSteps. (b) shows BE 1 final throughput (boxplot, five runs) varying ExperimentSteps; BE 2 has near zero final throughput in all cases.*

ExperimentStep of at least 500MB/s. ExperimentStep also influences how quickly the Nyx QoS can ensure LC tail latency. As shown in Figure 5.15.a, a larger ExperimentStep indicates faster convergence. However, it increases the risk of over-throttling BE caches and lowering system utilization. ExperimentStep in Nyx QoS defaults to 500MB/s for good interference analysis and high system utilization while maintaining a reasonable convergence time.

Figure 5.16 shows how DonateStep affects Nyx proportional resource allocation. A larger DonateStep causes faster idle resource donation, but also potentially large performance fluctuations (e.g., 60% DonateStep, 12s and 18s in the figure). At runtime, cache throughput always varies slightly, causing donation adjustments. These adjustments are subtle with small DonateSteps but significant with large ones. The fluctuation harms donors by slowing them down at times (exceeding the limit). Nyx uses a 10% DonateStep, which balances between quick resource donation and steady donor performance.

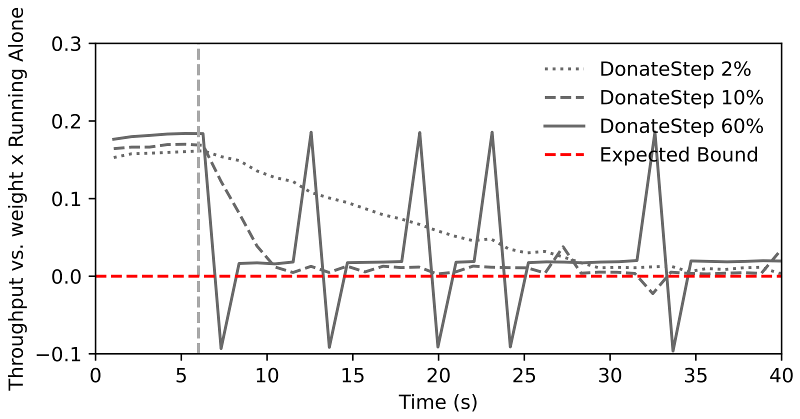


Figure 5.16: **Proportional Share: DonateStep Sensitivity Analysis.** *We use the same setup as in Figure* **??** *when B is write-heavy. This figure shows A, the donator's throughput over time with different DonateStep.*

5.4 Discussion

Beyond Basic Policies: Nyx can be extended to more sophisticated policies for more complex setups. For instance, a proportional sharing policy can be applied across groups of caches. Then, within a group, another sharing policy (e.g., QoS) can be enforced. We leave a full study as a future work.

Multi-tenant Caching Alternatives: Nyx manages caches, each with its own space. There are alternatives to shared caching; for instance, a single large instance can be shared by multiple users[199]. This model can make use of the Nyx resource usage accounting and interference analysis techniques. However, it may create new problems like: how should users be charged for PM writes to commonly cached objects? A full exploration is our future work.

Smarter Parameter Value Selection: i) Adaptive parameters can be beneficial, e.g., the DonateStep can be larger when it is far from the threshold (for quick donation) and smaller when it is close (to avoid performance fluctuations). ii) Auto-tuning[153, 221] may ease the load for choosing

parameter values. We leave these optimizations as future work.
Security: Nyx policies can be attackable, e.g., in resource limiting, an adversary client may limit its access in the first ticks while putting significant load in the last. A solution would be to use randomized measuring points rather than fixed ones. We leave Nyx security studies as future work.

Moreover, Nyx now assumes the PM access library within its trust boundary; this may not always be feasible. We leave future usage of dynamic instrumentation or hardware based solutions (for access regulation and resource usage accounting) as future work.

5.5 Conclusions

In this chapter, we demonstrated that prior DRAM or storage device-intended approaches for access regulation, resource-usage estimation, and interference analysis fail to work on PM due to PM's unique properties. As we summarized, these mechanisms are fundamental for popular sharing policies such as resource limiting, QoS-awareness, fair slowdown, and proportional sharing. Achieving PM sharing across multiple tenants hence become challenging. To address these challenges, we introduced Nyx, which enables these mechanisms in a lightweight manner without hardware support. Using profiled resource consumption knowledge of various access patterns, Nyx supports PM resource usage accounting. Nyx also supports cross-tenant interference analysis via runtime micro-experiments. We used multi-tenant key-value stores as a specific setup. We showed that Nyx mechanisms can support a variety of multi-tenant cache sharing policies, meeting performance or sharing goals better than earlier DRAM or storage approaches.

6 Related Work

In this chapter, we discuss how previous works relate to three parts of this dissertation. First, we discuss related works to our Optane SSD device characterizations (Section 6.1), such as previous PM studies, classic device characterization works, and real-world Optane SSD deployment experience. Next, we discuss how Non-Hierarchical Caching (NHC) differs from other related caching approaches (Section 6.2). Finally, we describe existing works on multi-tenant caching, PM caching, and PM interference (Section 6.3); we discuss how these works are insufficient for multi-tenant PM sharing.

6.1 Characterizing Modern PM Devices

Our Optane SSD device characterization is inspired by many prior device characterization studies.

Previous PM Studies: Previous PM studies have been available for years. These studies introduce variable PM materials [81, 158, 202] and PM device prototypes [1, 141]. These fundamental studies describe the properties of PM materials such as Phase-Change Memory, as well as the device engineering concepts needed to incorporate PM modules into real products. While many device internal details for Intel Optane devices (Optane

SSD and Optane DC PM) are never revealed, previous PM studies serve as the foundation for our investigations.

Yang et al. [151, 238] investigated Optane DC PM performance characteristics, a different type of PM device than the Optane SSD we studied. Our studies reveal both similar and radically different findings for these two devices. Both Optane SSD and Optane DC PM, for example, have significantly higher random access throughput and lower latency than traditional Flash SSDs. However, our study shows that Optane SSD has a minimum efficient access granularity of 4KB, whereas Optane DC PM has a granularity of 256B. Furthermore, we present that Optane SSD has the same read/write performance as well as the same random/sequential performance, whereas Optane DC PM has read/write asymmetry, and its random access performance is worse than its sequential access performance.

Finally, there are technical blogs [22, 40] that have covered interesting aspects of Optane SSDs. Unlike these blogs, our study was the first to thoroughly characterize Optane SSD performance and internals. Furthermore, during our investigation, these technical blogs were helpful in confirming some of our Optane SSD findings. For example, our experiments indicate that Optane SSD has internal parallelism degree of seven; this agrees with dismantling studies of Optane SSD [22]: the blog shows that Optane SSD has a controller connected to seven channels.

Classic Devices Characterizations: There have been numerous device characterizations for traditional devices. Schlosser and Ganger [208] presented the unique unwritten contract of HDDs. He et al. [143] summarized the unwritten contract of Flash-based SSDs. These previous unwritten contract studies prompted us to investigate how the unwritten contract for the new Optane SSDs should be structured.

Our fine-grained experiments to expose device internals are inspired by previous research on RAID-like device architecture analysis. Denehy et al. [122], for example, proposed algorithms for automatically detecting the number of disks, chunk size, and layout scheme in a RAID system. Similar detecting algorithms are presented by Chen et al. [110], but in the context of a NAND Flash SSD setup (which has a RAID-like internal architecture). In this study, we use and extend (for example, issuing concurrent accesses to 4KB blocks) these prior experimenting methodology, and present new findings on the Optane SSD device.

Real-World Optane SSD Deployment Experiences: Another body of work has investigated how to deploy applications on Optane SSDs. For instance, Optane SSDs are used as a caching layer between DRAM and Flash SSD in Facebook databases [129]. Facebook also stores embedding of trained neural networks on Optane SSDs [130]. These papers concentrate on part of the Optane SSD unwritten contract rules, such as “tiny access (less than 4KB) is inefficient." In contrast to them, we provide a comprehensive characterization of the devices. We also expose the Optane SSD’s internals to provide insight into the contract.

6.2 Evolving Caching for PM Hierarchies

The second part of our dissertation relates to existing works that manage various storage hierarchies.

Algorithms and Policies in Hybrid Storage Systems: Algorithms and policies for managing traditional hierarchy have been studied extensively [171, 181, 182, 184, 203, 222, 226, 236, 250]. Techniques have been introduced to optimize data allocation [14, 149, 205, 211, 222, 223], address translation [89, 207], identify hot data [15, 152, 155, 188, 191, 194, 204, 219, 226, 235] and perform data migration [97, 112, 131, 138, 184, 218, 226, 244]. Most previous work improves performance by focusing on workload access locality. In contrast, NHC improves by taking all devices and workloads into account.

Distributed Load Balancing and Multi-path Routing Techniques: Load balancing techniques in distributed systems [47, 58, 82] and multi-path routing [57, 172, 174, 185] have goals similar to our caching configuration. Multi-path routing [57, 172], for example, manages and utilizes multiple available paths for data transmission at the same time. Our work and multi-path routing both aim to maximize overall system bandwidth/performance by utilizing all available resources in network links/storage devices. Works for multi-path routing also considers other benefits such as fault tolerance and improved security, which we did not take into account in the caching setup. When compared to multi-path routing, our caching configuration presents unique challenges in terms of data placement and consistency. In contrast to network routing, where there is almost no limit to which path a request can take, Orthus can only do balancing if both devices have data replicas. As a result, Orthus must track cache hit rate and regularly increase hit rate to enable offloading. Orthus must also monitor dirty content in cache devices to ensure consistency.
Storage Optimization: A long line of pioneering work in storage management [83–85, 222, 226] shows how to trace workloads and optimize storage decisions for improved performance; NHC could fit into such a system, making short-term decisions to handle more dynamic workload changes, leaving longer-term optimization to a higher-level system.
Storage Aware Caching/Tiering: Our work shares aspects with storage-aware caching/tiering [80, 104, 134, 138, 145, 154, 160, 163, 164, 190, 232, 242], which considers more factors than hit rate. For instance, Oh et al. [190] propose over-provisioning in Flash to avoid the influence of SSD garbage collection. Modern devices like PM and Optane SSD have distinctive characteristics compared to Flash. We study the implications of these important emerging devices to caching/tiering.

BATMAN [113] shares a similar motivation to NHC: classic caching is not effective when the bandwidth of the capacity layer is a significant fraction of overall bandwidth. However, it investigates a much simpler hierarchy with fixed performance difference (4:1 between high-bandwidth memory and DRAM). Given the fixed difference, BATMAN splits cache accesses between HBM and DRAM statically. This approach would not work effectively on modern hierarchies where performance differences vary dynamically (e.g., depending upon the amount of writes or the level of parallelism in the workload).

Wu et al. [232] study tiering on SSDs and HDDs and recognize a similar problem: SSDs (or faster devices) can be the throughput bottleneck. To mitigate the problem, they proposed to periodically migrate data from SSDs to HDDs when the SSD response time is higher than that of HDDs. This approach is limited in three aspects. First, due to its tiering nature, it cannot react to workload changes quickly, its migration traffic can be significant, and it requires extra metadata to track objects across devices. Second, similar to SIB approach, it estimates workload intensity in a period and then migrates data based on the estimation; it hence struggles with dynamic workloads. Third, it is tuned for a specific hierarchy (SSDs and hard drives). Unlike this approach, NHC focuses on improving caching, adapts its behavior during runtime, can react to complex and dynamic workloads, and works well on a range of modern devices.

Managing PM-based Devices: Other related work incorporates PM-based devices into the memory-storage hierarchy for a variety of systems [79, 98, 141, 142, 146, 167, 183, 209].

This work includes extensive measurements for both Optane SSD [228] and Optane DC PM [151, 238]. New PM-based [200, 234] or low latency SSD-based [165, 173, 245] file systems and databases were also proposed. These systems are designed for single storage layer setup, as opposed to our work, which focuses on managing PM hierarchies.

Many works have evaluated the potential benefits of caching and tiering on PM. Kim et al. [159], for example, provide a simulation-based measurement of PM caching and tiering with performance numbers from a Micron all-PCM SSD prototype. The authors look into caching and tiering on a hierarchy of PM, Flash, HDD devices. The authors investigate not only absolute performance numbers but also cost-effectiveness (IOPS/$) when using PM caching and tiering. Their results show that PM devices show promising performance as a new storage layer in enterprise storage systems. Moreover, [133] provides a I/O cache simulator that assists the analysis of caching workloads on new storage hierarchies.

Strata [166] and Ziggurat [248] are file systems that tier data across a DRAM, PM, and SSD hierarchy. They propose various approaches for managing data movement between storage layers. Strata employed a log-structured scheme in which writes are first logged in higher layers before being batch moved to lower layers. Unlike Strata, Ziggurat uses an online profile of the application's access stream to predict the behavior of individual writes and direct them to different layers. In the background, Ziggurat estimates the "temperature" of files, and migrates the cold file data from PM to disks. Our NHC is compatible with these data movement schemes; in the meantime, NHC can improve the performance of these systems by enabling read-around, admission rejection, and dynamic feedback-based offloading.

Dulloor et al. [126], Arulraj et al.[87] and Zhang et al. [246] proposed PM-aware data placement strategies for the new storage hierarchy. Arulraj et al., for example, proposed that data movement between DRAM, PM, and SSD does not have to be strictly across neighboring layers, but can bypass the layer in the middle. For instance, data evictions from DRAM can bypass PM and end in SSD, saving PM writes. These strategies optimize data placement for PM hierarchies, which take effective in a longer period. NHC can work with them to provide further improvement by handling more dynamic workload changes; our techniques such as read around and admission rejection can take effect immediately in response to workload changes.

Finally, there have been many companies utilizing PM/ Optane SSD as a caching layer [106, 129, 130]. Our work is the first to analyze general caching and tiering on modern hierarchies through modeling and empirical evaluation. We are also the first to propose a generic solution (NHC) to realize the full performance benefits of such a hierarchy.

6.3 Evolving Sharing Mechanisms for PM

Finally, we discuss how previous works on multi-tenant caching, caching on PM, and PM interference studies are insufficient for multi-tenant PM sharing.

Multi-tenant in-mem key-value caching: Our work builds on past research in multi-tenant in-memory key-value cache systems. These efforts include techniques for allocating space across tenants [96, 114, 116, 199, 214] as well as optimization of individual cache instances [92, 94, 95, 115, 147, 187, 240]. Our work instead focuses on the challenges of access regulation and information extraction when many caches share PM.

PM Caching: There have been efforts to integrate PM with individual caching systems. Previous work covers databases [175, 229, 249], file systems [86, 166, 248], in-memory key-value caches [18, 64, 76], and general policies [93, 94]. However, to the best of our knowledge, we are the first to address PM issues in multi-tenant caching settings.

PM Interference: Several efforts have characterized PM devices [151, 224, 228, 238]. However, only a few have investigated the interference effect in PM. To our knowledge, Dicio [189] is the first work in this space. Both Dicio and our work observe the different read-write interference effect in PM.Dicio also provides a comprehensive measurement of PM-DRAM interference, which our work does not cover. Dicio's authors observe that concurrent DRAM and PM accesses can cause performance interferences;

it would be interesting to expand our work to include this new type of interference.

However, the goals of Dicio and Nyx differ. Dicio's purpose is to identify when PM DIMM bandwidth is saturated. Dicio approximates this by using the write pending queue (WPQ) delay as a heuristic. We, on the other hand, aim to provide mechanisms for per-client (not per-DIMM) resource usage accounting, slowdown estimation, and cross-client interference analysis. Dicio protects a single LC task from a single BE task, while our QoS policy applies to multiple clients. Dicio acknowledges that deciding which best-effort task to throttle, with PM media-level statistics, was challenging (and hence not done); we address this issue with a run-time method for interference analysis. Finally, Dicio extends Caladan [135] to use CPU scheduling to regulate PM accesses. This approach is applicable to all applications, including cache, but requires application modifications to use Caladan's unique runtime system (not fully Linux compatible). We leave CPU scheduling approaches for PM regulation to future work.

Sharing Other Resources: Efforts have been made to manage and share other resources such as network, CPU, LLC, storage devices, and locks [111, 135, 139, 140, 148, 150, 176, 195, 196, 198, 217]. They are essentially orthogonal to our work; we plan to integrate PM management into these systems in the future.

7 Conclusions

In this chapter, we first summarize each part of this dissertation (Section 7.1). Then, in Section 7.2, we discuss several high-level lessons we learned while working on this dissertation. In Section 7.3, we discuss potential future directions related to this dissertation. Finally, we discuss implications of Intel's recent decision to discontinue its Optane Memory business for our work (Section 7.4), and conclude (Section 7.5).

7.1 Summary

In this dissertation, we contribute to evolving the system stack for PM devices. This dissertation is comprised of three parts. In the first part, we characterized a new PM-based block device: the Intel Optane SSD. We formalize the rules (unwritten contract) that Optane SSD users must follow for optimal performance. In addition, we devise experiments to reveal the internals of the Optane SSD in order to provide insights for the unwritten contract. In the second part, we investigated how classic caching performs on modern storage hierarchies with PM devices, and demonstrated that the decades-old caching principle of maximizing hit rates is no longer sufficient today. We proposed Non-Hierarchical Caching, an enhanced caching approach for effectively utilizing capacity layer performance in modern storage hierarchies. Finally, we studied how

PM influence the effectiveness of existing sharing mechanisms designed for DRAM or block devices. We showed that existing mechanisms are insufficient for PM due to PM's unique characteristics. We also proposed the design of Nyx, which enables new sharing mechanisms to address PM sharing challenges. We now summarize each of these parts.

7.1.1 Characterizing Modern PM Devices

In the first part of this thesis, we characterized the performance of the new type of PM devices. We concentrated on the Intel Optane SSD, a popular PM-based block devices, because, at the time of our study, it was the only widely available PM option. We investigated Optane SSD performance in response to a variety of fine-grained access patterns (e.g., read vs. write, random vs. sequential, low vs. high parallelism, etc.).

Using our characterizations, we formalized a "unwritten contract" for Optane SSD users. The contract includes six critical rules for Optane SSD users to achieve optimal immediate performance, as well as one rule for sustained workloads relating to garbage collection in Optane SSDs. We also provided analyses to show how severe the impact can be when each contract rule is broken; we demonstrated that violating this contract can result in 11x worse read latency and limited throughput (only 20% of peak bandwidth), regardless of parallelism. We also designed micro-experiments to reveal the internals of Optane SSDs to better understand the insights of each unwritten rule. Internal parallelism, read-write scheduling mechanisms, block alignment granularity, and mapping policies from logical-block address (LBA) to physical-block address (PBA) in the Optane SSD are all experimentally revealed. We believe that our characterization of Optane SSD devices can serve as a foundation for future Optane SSD research.

7.1.2 Evolving Caching for PM Hierarchies

In the second part of the thesis, we analyzed classic caching on modern hierarchies, with PM devices filling the performance gap between DRAM and SSDs. We discovered that in modern storage hierarchies, the decades-old caching principle of maximizing hit rates is insufficient. In modern hierarchies, we observed a significant change: the performance difference between neighboring layers (e.g., DRAM/PM/Low-latency SSD/Flash SSD) is now much smaller than in traditional hierarchies (e.g. DRAM/HDD). Classic caching, which directs as many accesses as possible to the cache device, can leave significant available performance in capacity devices (e.g., PM) in such hierarchies unutilized.

To address this issue, we introduced Non-hierarchical Caching (NHC), an enhanced caching approach for maximizing aggregated cache and capacity layer performance out of modern hierarchies. NHC augments classic caching by adding three components: read around and admission rejection mechanisms, as well as a runtime feedback-based offloading policy. NHC's central idea is to enable access offloading from cache to capacity devices when the cache device is saturated, allowing NHC to utilize both cache and capacity device performance. We implemented NHC in both Orthus-CAS, a generic block-layer caching kernel module [108], and Orthus-KV, a user-level caching layer for an LSM-tree key-value store [168]. We demonstrated that NHC outperforms classic caching on various modern hierarchies (by up to 2x) under a variety of realistic workloads.

7.1.3 Evolving Sharing Mechanisms for PM

In the last part of the thesis, we addressed the question of how PM devices can be shared across multiple tenants. We focused on the specific configuration of multi-tenant memory-based look-aside key-value caches.

We began by summarizing the basic mechanisms (e.g., resource usage accounting, interference analysis) used to achieve various sharing goals (e.g., QoS-aware, proportional sharing). Then, we examined each sharing mechanism's issues on PM. We demonstrated that existing sharing mechanisms designed for DRAM/block devices do not readily translate to PM due to PM's unique characteristics such as 256B access granularity, asymmetric read/write performance, and severe and unfair interference between reads and writes.

To address this issue, we introduced NyxCache (Nyx), a standalone lightweight and flexible PM access regulation framework for multi-tenant key-value caches that is optimized for today's PM without special hardware support. Nyx's central contribution is a set of software mechanisms designed for PM to extract the information required to flexibly enforce popular sharing policies. We designed new mechanisms to efficiently i) regulate PM accesses, ii) obtain a client's PM resource usage, and iii) analyze inter-client interferences for PM. We then used these new mechanisms to revise key sharing policies on PM, including resource limiting, QoS, fair slowdown, and proportional sharing. We demonstrated the effectiveness of each sharing policy through various experimental studies.

7.2 Lessons Learned

In this section, we discuss a list of high-level lessons we learned while working on this dissertation.

7.2.1 Understanding Device Characteristics is Critical for Storage Research

Device characterization was important in all three parts of this dissertation (for example, characterizing Optane SSD in the first part, comparing

DRAM vs. PM vs. Low latency SSD vs. Flash in the second part, and characterizing Optane DC PM in the third part). We then evolved the system stack for PM based on our understanding of these devices. We learned that devices characterization is a necessary step in storage research.

According to our experience, there are three reasons why detailed device characterizations are important. First, characterization experiments allow us to have an accurate understanding of devices. In PM research, for example, PM materials have been described in classic papers; PM simulators/emulators have been built for years. However, as our Optane SSD unwritten contract study demonstrated, real devices can exhibit different characteristics than theoretical assumptions. When designing systems for real-world devices, these distinctions must be carefully considered. We can only gain an accurate understanding of real-world devices through characterization experiments.

Second, device specifications from manufacturers are frequently insufficient for device users. A device spec only includes performance numbers under limited cases (e.g., best cases). It cannot answer questions such as, “How will the device perform when applications require special access patterns?” (For example, in our Optane SSD study, access latency and throughput under high concurrency), “How will the device behave when multiple applications are running on the device at the same time?” (For example, read vs. write interference, as shown in our PM sharing study). To answer these questions, we need extensive micro-experiments.

Finally, characterizing devices forces us to model the internals of new devices qualitatively or quantitatively, as we did in the unwritten contract study. One significant benefit we discovered as a result of doing so is that it allows us to make reliable assumptions about which device features will be retained in the future and which may be completely changed. We believe that this knowledge is critical for new device research (e.g., PM); devices are still in development and can change at any time.

There are also lessons we learned about how to better characterize devices, which we will summarize here. First, comparison experiments, as we did in all of our studies, are usually very useful. By comparing one type of device (e.g., PM) to another (e.g., DRAM or Flash), we were able to learn a lot about the important differences between new and existing devices. Second, extreme or long-term characterizations can be useful and interesting at times. For instance, we used sustained write experiments in our Optane SSD characterization, and we studied PM performance under high concurrency in our PM characterization. They all produced surprising observations.

Overall, we believe that device characterization is an important step in the research of storage systems.

7.2.2 It is Helpful to Think Like a Novice at Some Point

One thing we've learned about research is that thinking like a novice can be useful at times. That is, at some point during our research, we must jump out all of the abstractions from previous works or summarized principles from previous studies/textbooks, think like a novice, and truly define what we want and expect from end users' perspectives.

Our PM caching study demonstrates the advantages of doing so. Maximizing hit rates is a decades-old well-known caching principle. It abstracts the desired end-to-end goal of maximizing performance from a storage hierarchy into maximizing the hit rate in the fast layer of the hierarchy. This abstraction has guided system designs for decades, but it also causes people to forget its underlying assumption: performance devices must be much much faster than capacity devices. Only by thinking like a novice, by recognizing what has been abstracted/ignored away, can we comprehend why the principle becomes insufficient in modern hierarchies with PM.

We believe that at some point during our system research, we should think like a novice and define: what does an end user expect from the system? how will you achieve the goal? and what is the relationship between your approach and well-known existing approaches? This procedure can be beneficial.

7.2.3 It is Important to Separate Mechanisms and Policies in System Research

We had a difficult time defining the problem of sharing on PM in the third part of this thesis. The difficulty stems from the fact that there are so many different sharing objectives in various configurations. In the Cloud, for example, the most popular sharing goal is to limit resource usage as the tenant paid. In a datacenter, proportional and quality-of-service-aware sharing are common goals. Diverged versions of these goals (for example, weighted proportional sharing) and the use of multiple policies are also common (e.g., proportional sharing across tenant groups and then resource limiting within each tenant group). With so many options, determining the problem with existing sharing techniques on PM devices was difficult.

It was the separation of fundamental mechanisms from the end-to-end goals (or policies) that solved our problems. We begin by studying popular sharing policies and defining the basic mechanisms required by them. And we discovered that, while there are numerous policies, only four common mechanisms are required (namely, request regulation, resource usage accounting, per-client slowdown estimation, and cross-client interference analysis). We were then able to define the problem with existing approaches on PM and improve them by focusing on the basic mechanisms.

We believe that distinguishing between mechanisms and policies is a useful skill for helping to define and solve system research problems.

7.3 Future Work

In this section, we discuss directions in which work done in this dissertation can be extended and directions that our research point to.

7.3.1 Energy Efficiency Characterization of PM Devices

In this thesis, we characterized performance aspects of new PM devices. With recent interest in datacenter energy consumption [30, 59], we believe a characterization of these devices in terms of energy efficiency (compared to DRAM and Flash) is an important future work.

There have been intriguing cases [13] demonstrating the potential of using PM to build energy-efficient memory systems (comparing to DRAM based). A thorough energy efficiency study that extends these works will be useful for future research.

Furthermore, we believe that a study of the implications of PM for large-scale distributed system energy efficiency is needed. If PM enables large memory capacity per server, network communication traffic may change; fewer network communications may be required because each server can now hold a larger portion of the overall dataset. Furthermore, PM-powered systems may necessitate fewer calls to remote services (e.g., Amazon S3). Reducing network traffic can result in significant savings in overall system power consumption. What implications do new PM devices have for distributed system overall energy efficiency? How can we use PM devices to increase the system energy efficiency? We believe there are promising energy-efficiency studies that can be investigated with PM devices.

7.3.2 Non-hierarchical Caching in the world of Memory/Storage Disaggregation

In Chapter 4, we designed and implemented non-hierarchical caching for single-node caching systems. Memory and storage disaggregation has recently been a radical development in memory/storage architecture. In such a configuration, applications can use not only local storage layers, but also remote layers (e.g., remote DRAM/PM, remote storage). We believe that a rethinking of caching is required in such a architecture, and that the non-hierarchical caching principle can also benefit caching in memory/storage disaggregation setups.

However, memory/storage disaggregation requires considerations that will necessitate further research. First, remote storage layers can have very different characteristics than raw storage devices (e.g., RDMA to access remote DRAM vs. access local DRAM). We need detailed characterization of these remote layers. Second, when comparing remote and local layers, we believe remote layers have greater diversity and complexity. Remote layers, for example, are distributed across the network. Congestion in networks, as well as different network speeds within and across data centers, can all change during runtime. More efforts are needed to manage hierarchies with remote layers. Third, computation offloading is a common feature of disaggregated memory or storage. Such hierarchies must be managed by taking into account not only storage accesses but also the computation associated with these accesses. Finally, today's storage hierarchies are becoming increasingly complex. How can such multi-layer hierarchies be effectively managed? Future research is required.

Overall, we believe that caching in the world of memory/storage disaggregation will necessitate significant future work for the system community.

7.3.3 Sharing with Better Hardware/Software Co-design

In Chapter 5, we examined the limitations of existing sharing mechanisms on PM and proposed software approaches to improve them. While this is an important first step, we believe it would be worthwhile to investigate how the PM hardware interfaces should be redesigned to enable PM sharing, what hardware functionalities should be added to PM, and how software and hardware should be codesigned to better realize sharing.

For example, while our cross-tenant interference analysis based on runtime micro-experiments works well for relatively stable workloads, this idea will be difficult to work if tenants change rapidly in milliseconds. Additionally, our software approaches assume a trusted software runtime across tenants, which is not always possible. It may be desirable to account for interference effects within the PM device. Important questions, we believe, include how future PM devices can expose an interface for querying the effect of interference across tenants, how to quantify interference in PM devices, and so on.

We believe that previous work on SSD [217, 245] or DRAM [186, 215] hardware-based sharing techniques can be inspiring for future work in this area.

7.3.4 Sharing of PM and DRAM pool in Disaggregated Memory Setup

In this thesis, we investigated the sharing of PM attached to a single node. Because PM and DRAM are promising building blocks for future disaggregated memory setups, the natural question is: how can we share the PM and DRAM pool among multiple tenants? In this configuration, a memory pool is made up of multiple memory nodes, each of which can contain DRAM or PM; clients access the memory pool via fast networks such as RDMA.

We believe there are some high-level questions that should be considered: i) How to allocate diverse memory pool resources (DRAM vs. PM) to tenants? ii) How to enforce tenant isolation across multiple memory nodes? We believe it is worthwhile to look into network switches and network interface cards (NIC) with extensive computing power today to enable DRAM/PM pool sharing.

7.4 Discussion: Demise of Intel Optane Memory Business

According to the Intel 2022 Q2 earning release [42], Intel starts to wind down its Optane Memory business. This announcement is a significant setback for the persistent memory community, as Intel Optane DC PM is the first and only commercially available PM device (as of today).

7.4.1 Implications for This Thesis

This news has a number of implications for our dissertation work. First, for our unwritten contract work on Optane SSDs (Chapter 3), our characterization of these specific devices may directly benefit fewer applications because they may need to migrate to other devices in the future. However, the experiments we designed to examine various aspects of the PM-based block devices, as well as the micro-experiments we designed to reveal the device's internals, will be useful for future PM-based block devices or low-latency SSDs. One general observation we made during our Optane SSD study is that better device specifications are required for applications to adapt to new devices. We believe that this lesson applies to future devices and may assist manufacturers in making devices that are easier to use in the future.

Second, we believe that our non-hierarchical caching work will continue to be useful for future PM devices or future storage hierarchies. We observe that the performance difference between neighboring layers in modern hierarchies is shrinking. This observation, we believe, will hold true for future high-performance new devices that bridge the gap between DRAM and Flash SSDs. Furthermore, we proposed general techniques for evolving classic caching. We make no assumptions about which devices are in the storage hierarchy. As a result, the non-hierarchical caching approach will continue to be useful for future storage hierarchies.

Finally, the mechanisms we proposed in the Nyx project, like non-hierarchical caching, will most likely be beneficial for future device sharing. For example, the idea of cross-client interference analysis based on runtime micro-experiments is applicable to all device types. And, we believe that when future memory devices become available, we will need to conduct a similar study of the fundamental sharing mechanisms as we have done in the Nyx project.

7.4.2 Implications of Our Thesis for Future PM Device Development

We believe that our thesis has implications for future PM device design. First, as demonstrated in Chapter 3, Flash-based SSDs now have significant maximum bandwidth (though the random access latency still suffer). And the price of Flash SSDs is very low and continues to fall today. So, in order to create a widely desired PM device, we believe that the PM device should still have copious internal parallelism (for high maximum bandwidth) and, most importantly, the price should be as low as possible.

Second, our caching study in Chapter 4 reveals that building efficient caching systems has become difficult in modern hierarchies (with so many different layers and the simple maximizing hit rate principle is not sufficient). Because caching is a critical use scenario for future PM devices. We believe that PM manufacturers should focus not only on device development but also on associated tiering or caching solutions.

Finally, our Optane DC PM sharing study in Chapter 5 shows that future PM devices should have more features in common with DRAM, so that existing DRAM sharing mechanisms can be used for PM. Alternatively, associated hardware sharing mechanisms must be developed as new devices are introduced.

Overall, we believe Optane DC PM (while winding down) continues to provide a wealth of experience for the system community in terms of future PM techniques. Furthermore, the work in this thesis will benefit future studies for future devices.

7.5 Closing Words

Data storage systems built from various storage devices have become indispensable in modern life. As the system community has done for decades, we must evolve the software system stack for each device type (tape, DRAM, HDD, Flash SSD, etc.). Although many efforts have been made for older devices, there have been few for PM-based ones. This dissertation contributes to evolving the system stack for PM. We came at it from three different perspectives: i) enhance understanding of PM device characteristics; ii) evolve caching for PM hierarchies; and iii) evolve sharing mechanisms for PM.

Through our studies, we presented i) how new real PM devices can have unexpected characteristics, ii) how decades-old principles (e.g., maximizing cache hit rates) may need to be rethought for PM, and iii) how existing systems (e.g., sharing mechanisms) must be reevaluated due to PM's unique characteristics. The findings we present suggest that the system community should reconsider how system stacks should be built around new PM devices. Our studies show that understanding how devices work, thinking like a novice at some point, and distinguishing mechanisms from policies are all critical for storage system research. Using the findings of our work, we demonstrated how new systems for PM (e.g., caching, sharing) can be proposed. We hope that our thesis will prompt other researchers to evolve other aspects of system stack for PM.

Our dissertation is a significant step towards better system stacks for PM, but it is only the beginning. We haven't looked into the energy efficiency of new PM devices. We haven't considered how to manage remote PM/DRAM layers in storage hierarchies, which may be common in future disaggregated memory architecture. We have also not looked into how to improve PM hardware interface designs for better PM sharing. We believe that the studies and ideas presented in this thesis will aid in future research to address some of these issues.

Bibliography

[1] 3D XPoint. https://en.wikipedia.org/wiki/3D_XPoint.

[2] 3D XPoint Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[3] Accelerate Ceph Clusters with Intel Optane DC SSDs. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/accelerate-ceph-clusters-with-optane-dc\-ssds-brief.pdf.

[4] Alibaba polardb. https://www.alibabacloud.com/product/polardb.

[5] Amazon cloud databases. https://aws.amazon.com/free/database/.

[6] Amazon elasticache pricing. https://aws.amazon.com/elasticache/pricing/.

[7] Amazon elasticache. https://aws.amazon.com/elasticache/.

[8] Apache spark. https://spark.apache.org/.

[9] Applications of artificial intelligence. https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence.

[10] Aws elasticache. https://aws.amazon.com/elasticache/redis/customers/.

[11] B+ tree. https://en.wikipedia.org/wiki/B%2B_tree.

[12] Budget fair queueing i/o scheduler. http://algo.ing.unimo.it/people/paolo/disk_sched/.

[13] Building An Ecosystem For Heterogenous Memory Supercomputing. https://www.nextplatform.com/2020/07/27/building-an-ecosystem-for-heterogeneous-memory-supercomputing/.

[14] Cache (computing). https://en.wikipedia.org/wiki/Cache_(computing).

[15] Cache replacement policies. https://en.wikipedia.org/wiki/Cache_replacement_policies.

[16] Caching and Tiering. https://storageswiss.com/2014/01/15/whats-the-difference-between\-tiering-and-caching/.

[17] Caching at reddit. https://redditblog.com/2017/1/17/caching-at-reddit/.

[18] Caching on pmem: an iterative approach. yue yao. https://www.snia.org/educational-library/caching-pmem-iterative-approach-2020.

[19] Ceph file system. https://docs.ceph.com/en/quincy/cephfs/.

[20] Cloud databases. https://en.wikipedia.org/wiki/Cloud_database.

[21] Computer data storage. https://en.wikipedia.org/wiki/Computer_data_storage.

[22] Decompsition of Intel Optane SSD 900P. https://www.anandtech.com/show/12136/the-intel-optane-ssd-900p-480gb-review.

[23] Dynamic random-access memory (dram). https://en.wikipedia.org/wiki/Dynamic_random-access_memory.

[24] File systems. https://en.wikipedia.org/wiki/File_system.

[25] Flash memory. https://en.wikipedia.org/wiki/Flash_memory.

[26] Google cloud databases. https://cloud.google.com/products/databases.

[27] Google memcache resource limit. https://cloud.google.com/appengine/docs/standard/python/memcache.

[28] Hard disk drive. https://en.wikipedia.org/wiki/Hard_disk_drive.

[29] High bandwidth memory. https://en.wikipedia.org/wiki/High_Bandwidth_Memory.

[30] HotCarbon: Workshop on Sustainable Computer Systems Design and Implementation. https://hotcarbon.org/.

[31] IMDT Use Case, Memcached. https://www.intel.com/content/www/us/en/support/articles/000026359/memory-and-storage/data-center-ssds.html.

[32] IMDT Use Case, Redis. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/imdt-solution-brief-in-memory-data-store.pdf.

[33] IMDT Use Case, Spark. https://www.intel.com/content/www/us/en/software/apache-spark-optimization-technology-brief.html.

[34] Intel mba issue with pm. https://github.com/intel/intel-cmt-cat/issues/170.

[35] Intel memory bandwidth allocation (mba). https://software.intel.com/content/www/cn/zh/develop/articles/introduction-to-memory-bandwidth-allocation.html.

[36] Intel Memory Drive Technology. https://www.intel.com/content/www/us/en/software/intel-memory-drive-technology.html.

[37] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[38] Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[39] Intel Optane SSD. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html.

[40] Intel Optane SSD 905P. https://www.tomshardware.com/reviews/intel-optane-ssd-905p,5600-2.html.

[41] Intel SSD 520 Series. https://ark.intel.com/content/www/us/en/ark/products/series/66202/intel-ssd-520-series.html.

[42] Intel winding down its optane memory business. https://www.forbes.com/sites/tomcoughlin/2022/07/28/intel-winding-down-its-optane-memory-business/?sh=5226545745b8.

[43] I/o scheduling. https://en.wikipedia.org/wiki/i/o_scheduling.

[44] ios file systems. https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html.

[45] ipmctl mediareads, mediawrites. https://docs.pmem.io/ipmctl-user-guide/instrumentation/show-device-performance.

[46] Linux block layer statistics. https://www.kernel.org/doc/Documentation/block/stat.txt.

[47] Load balancing (computing). https://en.wikipedia.org/wiki/Load_balancing_(computing).

[48] Log-structured merge-tree. https://en.wikipedia.org/wiki/Log-structured_merge-tree.

[49] Machine learning pipeline deployment and architecture. https://www.xenonstack.com/blog/machine-learning-pipeline.

[50] Magnetic tape. https://en.wikipedia.org/wiki/Magnetic_tape.

[51] Memcached Exstore. https://memcached.org/blog/nvm-caching/.

[52] Memcached. https://memcached.org/.

[53] Memcached stats. https://docs.oracle.com/cd/E17952_01/mysql-5.6-en/ha-memcached-stats.html.

[54] Memcachier. https://www.memcachier.com/.

[55] Micron Heterogeneous-Memory Storage Engine. https://www.micron.com/products/advanced-solutions/heterogeneous-memory-storage-engine.

[56] Micron X100 NVMe SSD. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology/x100.

[57] Multipath routing. https://en.wikipedia.org/wiki/Multipath_routing.

[58] Nginx and the power of two choices load-balancing algorithm. https://web.archive.org/web/20191212194243/https://www.nginx.com/blog/nginx-power-of-two-choices-load-balancing-algorithm/.

[59] NSF Call For Proposals: Design for Sustainability in Computing. https://www.nsf.gov/pubs/2022/nsf22060/nsf22060.jsp.

[60] Pci express. https://en.wikipedia.org/wiki/PCI_Express.

[61] Pelikan - twitter. https://twitter.github.io/pelikan/.

[62] Pelikan cache - taming tail latency and achieving predictability. https://twitter.github.io/pelikan/2020/benchmark-adq.html.

[63] Persistent memory. https://en.wikipedia.org/wiki/Persistent_memory.

[64] Pmem redis. https://github.com/pmem/pmem-redis.

[65] Redis enterprise cloud. https://redis.com/redis-enterprise-cloud/overview/.

[66] Redis. https://redis.io/.

[67] Remote direct memory access . https://en.wikipedia.org/wiki/Remote_direct_memory_access.

[68] RocksDB. https://rocksdb.org.

[69] Samsung 970 Pro. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/.

[70] Samsung 980 Pro Flash SSD. https://www.anandtech.com/show/15352/ces-2020\-samsung-980-pro-pcie-40-ssd-makes\-an-appearance.

[71] Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[72] Serial ata bus. https://en.wikipedia.org/wiki/Serial_ATA.

[73] Solid-state drive. https://en.wikipedia.org/wiki/Solid-state_drive.

[74] Top databases used in machine learning projects. https://analyticsindiamag.com/top-databases-used-in-machine-learning-projects/.

[75] Univac i. https://en.wikipedia.org/wiki/UNIVAC_I.

[76] The volatile benefit of persistent memory - memcached. https://memcached.org/blog/persistent-memory/.

[77] What is persistent memory. https://www.netapp.com/data-storage/what-is-persistent-memory/.

[78] SDC2020: Caching on PMEM: an Iterative Approach. https://www.youtube.com/watch?v=lTiw4ehHAP4, 2020.

[79] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-Mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985. ACM, 2019.

[80] Saba Ahmadian, Reza Salkhordeh, and Hossein Asadi. Lbica: A load balancer for i/o cache architectures. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1196–1201. IEEE, 2019.

[81] Ameen Akel, Adrian M Caulfield, Todor I Mollov, Rajesh K Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. *HotStorage*, 1:1, 2011.

[82] Ali M Alakeel et al. A guide to dynamic load balancing in distributed computer systems. *International Journal of Computer Science and Information Security*, 10(6):153–160, 2010.

[83] Guillermo A Alvarez, Elizabeth Borowsky, Susie Go, Theodore H Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems (TOCS)*, 19(4):483–518, 2001.

[84] Eric Anderson, Michael Hobbs, Kim Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. In *FAST '02*, Monterey, CA, January 2002.

[85] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems (TOCS)*, 23(4):337–374, 2005.

[86] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027, 2020.

[87] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv preprint arXiv:1901.10938*, 2019.

[88] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[89] Shi Bai, Jie Yin, Gang Tan, Yu-Ping Wang, and Shi-Min Hu. Fdtl: a unified flash memory and hard disk translation layer. *IEEE Transactions on Consumer Electronics*, 57(4):1719–1727, 2011.

[90] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[91] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *OSDI '10*, Vancouver, BC, December 2010.

[92] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[93] Nathan Beckmann, Phillip B Gibbons, Bernhard Haeupler, and Charles McGuffey. Writeback-aware caching. In *Symposium on Algorithmic Principles of Computer Systems*, pages 1–15. SIAM, 2020.

[94] Nathan Beckmann, Phillip B Gibbons, and Charles McGuffey. Block-granularity-aware caching. In *Proceedings of the 33rd ACM*

Symposium on Parallelism in Algorithms and Architectures, pages 414–416, 2021.

[95] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[96] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, 2018.

[97] Swapnil Bhatia, Elizabeth Varki, and Arif Merchant. Sequential prefetch cache sizing for maximal hit rate. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 89–98. IEEE, 2010.

[98] Timothy Bisson and Scott A Brandt. Flushing policies for nvcache enabled hard disks. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 299–304. IEEE, 2007.

[99] Daniel Bittman, Darrell DE Long, Peter Alvaro, and Ethan L Miller. Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[100] Christian Black, IT Michael Mesnier, and Terry Yoshii. Solid-State Drive Caching with Differentiated Storage Services. *Intel White Paper*, 2012.

[101] Simona Boboila and Peter Desnoyers. Performance Models of Flash-based Solid-State Drives for Real Workloads. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2011.

[102] Luc Bouganim, Björn Thór Jónsson, Philippe Bonnet, et al. uFLIP: Understanding Flash IO Patterns.

[103] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 281. Prentice Hall Upper Saddle River, 2003.

[104] Nathan C Burnett, John Bent, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2002.

[105] Geoffrey W Burr, Bülent N Kurdi, J Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, 2008.

[106] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[107] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[108] Open CAS. Open Cache Acceleration Software. `https://open-cas.github.io/`.

[109] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, V Nikitin, X Tang, S Watts, S Wang, et al. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics*, 46(6):1873–1878, 2010.

[110] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing. pages 266–277.

[111] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[112] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 379–392, 2016.

[113] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems*, pages 268–280, 2017.

[114] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[115] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory

caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[116] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[117] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. pages 143–154.

[118] Craciunas, Silviu S and Kirsch, Christoph M and Röck, Harald. I/o resource management through system call scheduling. *ACM SIGOPS Operating Systems Review*, 42(5):44–54, 2008.

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

[120] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. Qos-aware admission control in heterogeneous datacenters. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 291–296, 2013.

[121] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[122] Timothy E Denehy, John Bent, Florentina I Popovici, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Deconstructing

Storage Arrays. In *ACM SIGARCH Computer Architecture News*, volume 32. ACM, 2004.

[123] Peter Desnoyers. Empirical Evaluation of NAND Flash Memory Performance. *ACM SIGOPS Operating Systems Review*, 44(1), 2010.

[124] Peter Desnoyers. What Systems Researchers Need to Know about NAND Flash. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

[125] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. pages 1–8.

[126] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 15. ACM, 2016.

[127] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335–346, 2010.

[128] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[129] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon,

and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[130] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-volatile Memory for Storing Deep Learning Models. *arXiv preprint arXiv:1811.05922*, 2018.

[131] Ahmed Elnably, Hui Wang, Ajay Gulati, and Peter J Varman. Efficient qos for multi-tiered storage systems. In *HotStorage*, 2012.

[132] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-Scale Graph Processing on Emerging Storage Devices. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[133] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. Desperately seeking... optimal multi-tier cache configurations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[134] Brian Forney, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2002.

[135] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[136] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[137] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *arXiv preprint arXiv:1904.07162*, 2019.

[138] Jorge Guerra, Himabindu Pucha, Joseph S Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, volume 11, pages 20–20, 2011.

[139] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, volume 9, pages 85–98, 2009.

[140] Ajay Gulati, Arif Merchant, and Peter J Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI*, volume 10, pages 437–450, 2010.

[141] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9), 2017.

[142] Theodore R Haining and Darrell DE Long. Management policies for non-volatile write caches. In *1999 IEEE International Performance, Computing and Communications Conference (Cat. No. 99CH36305)*, pages 321–328. IEEE, 1999.

[143] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017.

[144] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[145] David A Holland, Elaine Angelino, Gideon Wald, and Margo I Seltzer. Flash caching on the storage client. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 127–138, 2013.

[146] Morteza Hoseinzadeh. A survey on tiering and caching in high-performance storage systems. *arXiv preprint arXiv:1904.11560*, 2019.

[147] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, 2015.

[148] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, 2017.

[149] Ilias Iliadis, Jens Jelitto, Yusik Kim, Slavisa Sarafijanovic, and Vinodh Venkatesan. Exaplan: queueing-based data placement and provisioning for large tiered storage systems. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 218–227. IEEE, 2015.

[150] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.

[151] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Sub-

ramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[152] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 327–337. IEEE, 2003.

[153] Yichen Jia and Feng Chen. Kill two birds with one stone: Auto-tuning rocksdb for high bandwidth and low latency. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 652–664. IEEE, 2020.

[154] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4, pages 8–8, 2005.

[155] Shudong Jin and Azer Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 254–261. IEEE, 2000.

[156] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[157] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Caliper: Interference estimator for multi-tenant environments sharing architectural resources. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–25, 2019.

[158] Takayuki Kawahara. Scalable Spin-transfer Torque RAM Technology for Normally-off Computing. *IEEE Design & Test of Computers*, 28(1):52–63, 2010.

[159] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 33–45, 2014.

[160] Jaehyung Kim, Hongchan Roh, and Sanghyun Park. Selective i/o bypass and load balancing method for write-through ssd caching in big data analytics. *IEEE Transactions on Computers*, 67(4):589–595, 2017.

[161] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. MASCOTS '11, 2011.

[162] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A Simulator for Nand Flash-based Solid-State Drives.

[163] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 45–58, 2013.

[164] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing*, pages 51–60. IEEE, 2015.

[165] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast {NVM} storage with udepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[166] 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*. ACM, 2017.

[167] Chun-Hao Lai, Jishen Zhao, and Chia-Lin Yang. Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[168] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. pages 133–148.

[169] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[170] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage.

[171] Donghee Lee, Jongmoo Choi, Jun-Hum Kim, Sam H. Noh, Sang Lyul Min, Yookum Cho, and Chong Sang Kim. On The Existence Of A Spectrum Of Policies That Subsumes The Least Recently Used (LRU) And Least Frequently Used (LFU) Policies. In *SIGMETRICS '99*, Atlanta, GA, May 1999.

[172] S-J Lee and Mario Gerla. Split multipath routing with maximally disjoint paths in ad hoc networks. In *ICC 2001. IEEE international conference on communications. Conference record (Cat. No. 01CH37240)*, volume 10, pages 3201–3205. IEEE, 2001.

[173] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[174] Haejung Lim, Kaixin Xu, and Mario Gerla. Tcp performance over multipath routing in mobile ad hoc networks. In *IEEE International Conference on Communications, 2003. ICC'03.*, volume 2, pages 1064–1068. IEEE, 2003.

[175] Gang Liu, Leying Chen, and Shimin Chen. Zen: a high-throughput log-free oltp engine for non-volatile main memory. *Proceedings of the VLDB Endowment*, 14(5):835–848, 2021.

[176] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[177] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. hStorage-DB: Heterogeneity-aware Data Management to Exploit the Full Capability of Hybrid Storage Systems. *Proceedings of the VLDB Endowment*, 5(10), 2012.

[178] 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)*, 2017.

[179] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[180] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[181] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST '03*, San Francisco, CA, April 2003.

[182] Michael Mesnier, Feng Chen, Tian Luo, and Jason B Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 57–70. ACM, 2011.

[183] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2015.

[184] David Montgomery. Extent migration scheduling for multi-tier storage architectures, November 5 2013. US Patent 8,578,107.

[185] Asis Nasipuri, Robert Castaneda, and Samir R Das. Performance of multipath routing for on-demand protocols in mobile ad hoc networks. *Mobile Networks and applications*, 6(4):339–349, 2001.

[186] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222. IEEE, 2006.

[187] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *NSDI '13*, pages 385–398, Lombard, IL, April 2013.

[188] Junpeng Niu, Jun Xu, and Lihua Xie. Hybrid storage systems: a survey of architectures and algorithms. *IEEE Access*, 6:13385–13406, 2018.

[189] Jinyoung Oh and Youngjin Kwon. Persistent Memory Aware Performance Isolation with Dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 97–105, 2021.

[190] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012.

[191] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[192] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *SIGMOD '93*, pages 297–306, Washington, DC, May 1993.

[193] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[194] Dongchul Park and David HC Du. Hot Data Identification for Flash-Based Storage Systems Using Multiple Bloom Filters.

[195] Jinsu Park, Seongbeom Park, and Woongki Baek. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[196] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: a hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[197] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, volume 12, pages 13–13, 2012.

[198] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[199] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. Fairride: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, 2016.

[200] Sheng Qiu and AL Narasimha Reddy. Nvmfs: A hybrid file system for improving random write in nand-flash ssd. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2013.

[201] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (Third Edition)*. McGraw-Hill, 2004.

[202] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[203] Benjamin Reed and Darrell DE Long. Analysis of caching algorithms for distributed file systems. *ACM SIGOPS Operating Systems Review*, 30(3):12–21, 1996.

[204] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement.

[205] Reza Salkhordeh, Hossein Asadi, and Shahriar Ebrahimi. Operating system level data tiering using online workload characterization. *The Journal of Supercomputing*, 71(4):1534–1562, 2015.

[206] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 267–280. ACM, 2012.

[207] Andre Schaefer and Matthias Gries. Adaptive address mapping with dynamic runtime memory mapping selection, 2012. US Patent 8,135,936.

[208] Steven W. Schlosser and Gregory R. Ganger. MEMS-based Storage Devices and Standard Disk Interfaces: A Square Peg in a Round Hole? In *FAST '04*, San Francisco, CA, April 2004.

[209] Priya Sehgal, Sourav Basu, Kiran Srinivasan, and Kaladhar Voruganti. An empirical study of file systems on nvm. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[210] Kai Shen and Stan Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 67–78, 2013.

[211] Haixiang Shi, Rajesh Vellore Arumugam, Chuan Heng Foh, and Kyawt Kyawt Khaing. Optimal disk storage allocation for multitier storage system. *IEEE Transactions on magnetics*, 49(6):2603–2609, 2013.

[212] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.

[213] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[214] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181. ACM, 2015.

[215] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 62–75. IEEE, 2015.

[216] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650. IEEE, 2013.

[217] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 397–410. IEEE, 2018.

[218] Elizabeth Varki, Allen Hubbe, and Arif Merchant. Improve prefetch performance by splitting the cache replacement queue. In *IEEE International Conference on Advanced Infocomm Technology*, pages 98–108. Springer, 2012.

[219] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[220] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance Insulation for Shared Storage Servers. In *FAST*, volume 7, pages 5–5, 2007.

[221] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-tuning query scheduling for analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1879–1891, 2021.

[222] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, 2017.

[223] Hui Wang and Peter Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceed-*

ings of the 12th USENIX Conference on File and Storage Technologies (FAST 14), pages 229–242, 2014.

[224] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508. IEEE, 2020.

[225] Wikipedia. ext4. en.wikipedia.org/wiki/Ext4, 2008.

[226] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[227] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012.

[228] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane ssd. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). USENIX Association, Renton, WA*, 2019.

[229] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323, 2021.

[230] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-

Dusseau, and Remzi H. Arpaci-Dusseau. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323, 2021.

[231] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {NyxCache}: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, 2022.

[232] Xiaojian Wu and AL Narasimha Reddy. A novel approach to manage a hybrid storage system. *JCM*, 7(7):473–483, 2012.

[233] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st international symposium on high performance computer architecture (HPCA)*, pages 476–488. IEEE, 2015.

[234] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[235] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Fast*, volume 7, pages 25–25, 2007.

[236] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative Caching with Return on Investment. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2013.

[237] Gala Yadgar and Moshe Gabel. Avoiding the Streetlight Effect: I/O Workload Analysis with SSDs in Mind.

[238] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. *arXiv preprint arXiv:1908.03583*, 2019.

[239] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

[240] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *NSDI*, pages 503–518, 2021.

[241] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T Kaushik, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 474–489, 2015.

[242] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. Autotiering: automatic data placement manager in multi-tier all-flash datacenter. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2017.

[243] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

[244] Gong Zhang, Lawrence Chiu, and Ling Liu. Adaptive data migration in multi-tiered storage based cloud environment. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 148–155. IEEE, 2010.

[245] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI 18*), pages 477–492, 2018.

[246] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, pages 1–27, 2020.

[247] Jishen Zhao, Onur Mutlu, and Yuan Xie. Firm: Fair and high-performance memory control for persistent memory systems. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–165. IEEE, 2014.

[248] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies* (*FAST 19*), pages 207–219, 2019.

[249] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2195–2207, 2021.

[250] Yuanyuan Zhou, James F. Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX '01*, pages 91–104, Boston, MA, June 2001.

[251] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI 18*), 2018.