Optimizing Modern Heterogeneous Systems: From Storage to Memory

By

Kaiwei Tu

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: 05/21/2026

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

Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Michael Swift, Professor, Computer Sciences
Ming Liu, Assistant Professor, Computer Sciences
Dan Negrut, Professor, Mechanical Engineering

© Copyright by Kaiwei Tu 2026
All Rights Reserved

To everyone who believed in me.

Acknowledgments

First and foremost, I would like to thank my advisors, Andrea and Remzi, for their careful guidance and support throughout my Ph.D. journey. They are my role models as systems researchers: dedicated and passionate about systems research. Their spirit has shaped me into a better researcher. Andrea is thoughtful about every aspect of my projects, from the technical details to the writing and presentation, and her dedication to perfection drove me to polish each piece of work. Remzi's vision and ability to generalize a problem always lit the way when a project was still in chaos and guided me toward the right research problems. More importantly, they care deeply about their students: they gave me the freedom to explore what I am interested in, encouraged me to take on challenges, and provided thoughtful feedback on my research. I could not ask for better advisors. Thank you, Andrea and Remzi!

I also want to thank the rest of my committee: Michael Swift, Ming Liu, and Dan Negrut. Michael and Ming also served on my preliminary exam committee, where they posed questions that prompted me to think more deeply about my work on heterogeneous storage and memory systems. I thank Dan for his willingness to serve on my committee; his GPU programming course, which I took early in my Ph.D., was a fascinating introduction to high-performance computing. Thank you, Michael, Ming, and Dan.

I am grateful to have worked with a group of smart colleagues at UW–Madison, including Kan Wu, Chenhao Ye, Shawn Zhong, Suyan Qu, Anthony Rebello, Guanzhou Hu, Tingjia Cao, Yifan Dai, Jing Liu, Sambhav Satija, Xiangpeng Hao, John Shawger, Ram Alagappan, Aishwarya Ganesan, Yuvraj

Patel, Kai Mast, Vinay Banakar, and many others. I was especially fortunate to work with Kan on the Orthus project early in my Ph.D., a project that inspired the two works that followed. Over the three projects we worked on together, I learned a lot from him about how to break down a research problem and how to design experiments that present the findings clearly. It was always a pleasure to work and discuss research ideas with him. I am also grateful to everyone in the group for the time we shared inside and outside the lab, for the feedback during practice talks, and for the conversations about research, life, and everything beyond. Beyond research, I would also like to thank my friends in Madison. Thank you all for everything we shared throughout this journey: the basketball, the card games, the dinners, and so much more. It was those moments that made this journey unique and meaningful.

Finally, I want to thank my family: my parents, for their love, support, and belief in me. I would not be where I am today without them. I also cannot express how grateful I am to Zhongling, my girlfriend, who has stood by me throughout this journey. Together we raise our two adorable Ragdoll cats, Cookie and Puff. Zhongling is an excellent researcher herself, working on theoretical machine learning, and her passion for life brightened mine, even in the darkest moments. It was you who made this long journey feel short and bright. Thank you all for your love.

Contents

Acknowledgments **ii**

Contents **iv**

List of Tables **viii**

List of Figures **x**

Abstract **xvi**

1 Introduction **1**

1.1 Optimizing Caching for Modern Non-Hierarchical Storage . . . 4

1.2 Getting the MOST with Mirror-Optimized Storage Tiering . . . 6

1.3 Mapping-Based Balancing in Non-Hierarchical Memory 9

1.4 Contributions and Highlights 11

1.5 Outline of Chapters 13

2 Heterogeneous Storage and Memory Systems **16**

2.1 CXL Devices 17

2.2 Persistent Memory 19

2.3 Low-latency SSDs 20

2.4 Conclusion 21

3 Optimizing Caching for Modern Non-Hierarchical Storage **22**

3.1 Background 23
3.1.1 Motivation 23
3.1.2 Non-Hierarchical Caching 25
3.1.3 Previous Cache-Bypass Approaches 30
3.2 Implementation 34
3.2.1 Open CAS and Orthus-CAS 34
3.2.2 Baseline Implementations 36
3.3 Evaluation 37
3.3.1 Comparison to Previous Systems 37
3.3.2 Target Performance Metrics 38
3.4 Conclusion 39

4 Getting the MOST with Mirror-Optimized Storage Tiering 40
4.1 Motivation and Background 41
4.1.1 Multi-Device Hierarchies 42
4.1.2 Existing Approaches 43
4.1.3 Storage vs. Memory Hierarchies 47
4.2 Mirror-Optimized Storage Tiering 48
4.2.1 Basic Design 49
4.2.2 Design Details 49
4.2.3 Implementation: Cerberus 55
4.3 Evaluation 59
4.3.1 Comparison to Previous Approaches 59
4.3.2 Dynamic Workloads 62
4.3.3 Cerberus In-depth Analysis 66
4.3.4 CacheLib 68
4.4 Discussion 74
4.5 Conclusion 75

5 Mapping-Based Balancing in Non-Hierarchical Memory 76

5.1 Motivation and Background 77
5.1.1 Load Balancing in Heterogeneous Memory 77
5.1.2 Why Migration-Based Balancing is Insufficient 79
5.1.3 Opportunity: Mapping-Based Load Balancing 83
5.2 Hecaton Design 84
5.2.1 Overview 85
5.2.2 Fast Load Balancing 86
5.2.3 Mirror Maintenance 88
5.2.4 Identifying Useful Mirrors 90
5.2.5 Implementation 91
5.3 Evaluation 93
5.3.1 Convergence and Interference Under Contention 94
5.3.2 Sensitivity and Graceful Degradation 97
5.3.3 Tracking Bursty Contention 99
5.3.4 Realistic Memory-Intensive Workloads 103
5.4 Conclusion 105

6 Related Work **107**
6.1 Optimizing Caching for Modern Non-Hierarchical Storage . . 107
6.2 Mirror-Optimized Storage Tiering 109
6.3 Heterogeneous Memory System 112

7 Conclusion and Future Work **114**
7.1 Summary 114
7.1.1 Optimizing Caching for Modern Non-Hierarchical Storage 115
7.1.2 Mirror-Optimized Storage Tiering 116
7.1.3 Mapping-Based Load Balancing 117
7.2 Future Work 117

7.2.1 Sustainability in Heterogeneous Storage System with High-Capacity SSD . . . 117
7.2.2 Performance Isolation in Heterogeneous Storage and Memory Systems . . . 119
7.2.3 Optimizing the Software Stack for Hybrid Devices . . . 120
7.3 Lessons Learned . . . 121
7.3.1 Measure First, then Build . . . 121
7.3.2 Reason with Mechanisms, then Policies . . . 121
7.3.3 Start with Strict Assumptions, then Relax . . . 122
7.4 Closing Remarks . . . 123

Bibliography **125**

List of Tables

2.1 **Performance of Representative Devices.** *DRAM, Micron DDR5-8000 RDIMM [83]; CXL memory, CXL-D Type 3 DRAM memory expander (CXL 1.1, PCIe 5.0 ×16, 4×DDR5); Persistent memory, Intel Optane PMem 200 [51]; CXL.mem + SSD, Samsung CMM-H prototype; Low-latency SSD, Intel Optane SSD P5800X [52]; Flash SSD (PCIe 3.0), Micron 9300 PRO/MAX [82]; Flash SSD (PCIe 4.0), Samsung 990 PRO [102]; Flash SSD (PCIe 5.0), Micron 9550 PRO/MAX [84]; SATA SSD, Samsung PM893 [104]; Hard disk, WD Ultrastar DC HC690 (7200 RPM) [119]. Throughput is per device/module unless otherwise stated.* 18

3.1 **Diversified Storage Devices.** *Representative latency, bandwidth, and cost for modern storage and memory devices. Data taken from SK Hynix DRAM (DDR4, 16 GB), Intel Optane DCPM, low-latency SSDs (Intel Optane SSD 905P, Micron X100), NVMe Flash SSD (Samsung 970 Pro), and SATA Flash SSD (Intel 520 SSD); low-latency and NVMe Flash SSDs assume PCIe 3.0. Bandwidth and latency differences between adjacent rows are now within an order of magnitude, breaking the assumption that the hierarchy is strict.* 24

3.2 **Different Target Metrics.** *Orthus-CAS performance when configured with throughput, average latency, or P99 tail latency as the target metric. Bold marks the best Orthus-CAS result for each metric in each hierarchy.* 39

4.1 **Device Performance.** *Latency measured for a single-thread read workload; bandwidth for a 32-thread workload. Remote device is connected via a 25 Gbps link.* 42

4.2 **Qualitative Comparison of Different Techniques in a Modern Storage Hierarchy.** *Bandwidth utilization is categorized as follows: Low means no load-balancing mechanism; Medium represents limited load-balancing with restrictions under specific workloads; High indicates effective load-balancing across a wide range of workloads.* 44

4.3 **In-Memory Metadata per Segment.** 58

4.4 **Production Trace Distributions from CacheBench.** *Flat-kvcache and graph-leader are from an application cache with small value sizes, resulting in mostly random traffic; kvcache-reg and kvcache-wc are from a storage cache with large value sizes, leading to log-structured traffic. LoneGet and LoneSet are requests for a key not present in the cache.* . . 68

4.5 **Average and P99 GET Latency (ms) of Production Workloads.** . 72

5.1 **Comparison of Heterogeneous Memory Systems.** *Data Layout specifies whether the system maintains one or multiple copies of each page. Placement refers to the criterion the system uses to decide where to place data. For convergence and interference, Fast/Low are desirable, Slow/High are undesirable, and – indicates no load balancing.* 80

5.2 **Realistic workload characteristics.** *All workloads use 4 application threads except btree, which is single-threaded. Stack and other small, bounded-size kernel data structures are not managed by HeMem and are allocated in DRAM; their footprint is negligible.* 103

List of Figures

2.1 **Modern Heterogeneous Storage and Memory Architecture.** *Three new device classes—CXL memory, persistent memory (PM) DIMM, and low-latency SSD—bridge the performance gap between DRAM and SATA/NVMe flash SSD, narrowing the gap between neighboring tiers in the modern hierarchy.* 17

3.1 **Caching, Tiering, and Non-Hierarchical Caching.** *(a) Caching copies hot data to the performance device D_{hi} on miss and serves all hits from D_{hi}, with replacement traffic flowing $D_{hi} \rightarrow D_{lo}$. (b) Tiering splits access between D_{hi} and D_{lo} and migrates data between them in the background on long time scales. (c) Non-hierarchical caching keeps D_{hi} populated like classic caching but, when D_{hi} saturates, redirects a tunable fraction of hits to D_{lo} via the* `load_admit` *switch (dashed).* . . 24

3.2 **Non-Hierarchical Caching Architecture.** *NHC adds two decision points to a classic cache controller:* `data_admit` *controls whether a read miss promotes a block to D_{hi}, and* `load_admit` *controls the fraction of read hits served from D_{hi} versus redirected to D_{lo}. The scheduler monitors a target metric $f(X)$ and tunes both knobs in feedback. NHC is transparent to users; any classic caching implementation can be upgraded to be an NHC one.* 27

3.3 **SIB Architecture.** *The SIB module in the kernel block layer routes each process's reads either through the SSD write-through cache (regular path) or directly to an HDD (bypass). A saturation detector monitors SSD bandwidth (*BW*) and average await time (T_w); when the bandwidth z-score exceeds* 4 *and T_w is above its observed minimum, the SSD is declared saturated. SIB then selects entire processes from its LRU list of large, sequential I/O streams (here P_4 and P_6) and redirects their reads to idle HDDs. Selection is per-process: once a process is in the bypass set, all of its reads avoid the SSD until the process terminates.* 30

3.4 **LBICA Operation.** *LBICA periodically samples the cache and disk queues, then runs a three-stage pipeline.* ***Stage 1*** *detects whether the SSD cache is the bottleneck by comparing the queue time of the cache against that of the disk; queue time is defined as the number of pending requests in the queue multiplied by the device's average request latency, and the cache is declared the bottleneck when its queue time is larger than the disk's.* ***Stage 2*** *classifies the in-queue requests by their mix of read (R), write (W), promote (P), and evict (E) operations into one of four groups.* ***Stage 3*** *switches the cache write policy accordingly: Write-Only for random-read workloads, Read-Only for mixed read/write, Write-Back with tail bypass for write-intensive workloads, and plain Write-Back for sequential reads where the cache has no impact. LBICA does not redirect any read hits.* 32

3.5 **Open CAS Architecture.** *Open CAS is a kernel block-layer caching framework. User applications issue I/O through the VFS and a chosen logical file system. The Top adapter forwards block requests into the Open CAS Framework (OCF), which decides whether each request hits the cache, then dispatches to the Bottom adapter. The Bottom adapter splits I/O into two paths through the standard Linux block and device-driver layers: backend (blue) reaches the slower capacity device, and caching (orange) reaches the faster cache device. NHC's routing hook and scheduler live inside OCF.* 35

3.6 **NHC vs. SIB and LBICA.** *(a) Static read-only workload. (b) Dynamic workloads where the write ratio changes periodically (e.g., 10 s), varying randomly between 0% and 5%.* 38

4.1 **MOST Data Layout.** *Data are logically grouped but not physically placed together on the device.* 47

4.2 **MOST Architecture.** 50

4.3 **CacheLib Architecture.** *This figure shows CacheLib's architecture and the lookup workflow. A lookup first checks the DRAM cache (①) and immediately returns the object on a hit (②). On a miss, it checks the flash cache (③), issuing a read to the underlying storage devices (④a); if the object is found, it returns the result (④b). A flash cache hit promotes the item to DRAM and may evict an existing DRAM entry (⑤a, ⑤b). A miss in the flash cache (⑥) leads to a back-end access (⑦).* 56

4.4 **Static Workload.** *The workload is running on the Optane/NVMe hierarchy with 750 GB working set. In (a), under intensity 2.0×, Colloid, Colloid+, Colloid++, and Cerberus migrate 134 GB, 122 GB, 62 GB, and 50 GB of data in total. In (b), under intensity 2.0×, Colloid, Colloid+, Colloid++, and Cerberus migrate 0 GB, 93 GB, 80 GB, and 65 GB of data in total.* 60

4.5 **Dynamic Bursty Workload: Read-only.** *Running on Optane/NVMe hierarchy with 1.2 TB working set and same skewness as §4.3.1. Under the read-only workload, Colloid++ migrates 282 GB to the performance device and 262 GB to the capacity device. In comparison, Cerberus migrates 87 GB of data to the capacity device.* 63

4.6 **Dynamic Bursty Workload: Write-only and RW-mixed.** *Same setup as Figure 4.5. Colloid++ migrates 214 GB and 260 GB to the performance device and 186 GB and 239 GB to the capacity device under workloads (a) and (b), respectively. In comparison, Cerberus migrates 107 GB and 64 GB of data to the capacity device.* 64

4.7 **Limitation of Migration-based Approach.** *The same workload as Figure 4.5.* 65

4.8 **In-depth Analysis.** *In (a), workload is random read/write mixed (50% writes) at a high load (128 threads), varying the working set size, which is shown as a fraction of the total storage system capacity. The hotset is 20% of the working set with 90% access probability. In (d), the workload is a 256-thread workload with occasional write spikes every 0.1, 1, and 30 seconds.* 67

4.9 **Lookaside Cache Workload: Small Object Cache.** *The DRAM cache is restricted to 200 MB to stress the SOC in CacheLib. The workloads are 256 threads get/set-mixed Zipfian with 25M keys and 1 KB value. SOC is set to 100 GB, one third of the total capacity (100 GB/200 GB hierarchy).* 69

4.10 **Lookaside Cache Workload: Large Object Cache.** *Same setup as Figure 4.9, but the workloads are 256 threads get/set-mixed Zipfian with 5M keys and 16 KB value to stress the LOC.* 69

4.11 **Production Workloads.** *DRAM cache is set to 1 GB. For flat-kvcache and graph-leader, the SOC is set to one third of the total hierarchy capacity. We use 256 threads for flat-kvcache, graph-leader, and kvcache-wc and 80 threads for kvcache-reg. The cache throughput is normalized to HeMem's performance.* 71

4.12 **Dynamic Cache Workload.** *Read-heavy workload with 95% GET and 5% SET operations on Optane/NVMe hierarchy, where 20% of keys belong to the hotset; the hotset is accessed uniformly at random with a 90% probability. 25 million key-value pairs, with value sizes ranging from 2 KB to 4 KB; 1 billion operations. Size of SOC is 450 GB.* 73

4.13 **YCSB.** *CacheLib uses the default parameters from CacheBench with 4 GB DRAM cache. All the workloads are Zipfian ($\theta = 0.8$) with 1 KB values and 16-byte keys running under 256 threads with 20M records. The cache throughput is normalized to the throughput of the default (striping) system and the number above each bar is P99 latency in microseconds. Workload E is excluded due to CacheLib's lack of support for range queries.* 74

5.1 **Three approaches to load balancing.** *Migration-based balancing keeps a single copy and moves data when load shifts; it is applicable to both memory and storage systems. Routing-based balancing maintains two copies and routes each request to one of them; it is not applicable to memory systems. Mapping-based balancing maintains two copies and switches the page table mapping to redirect future accesses.* 79

5.2 **Load Burst.** *A 4-thread multichase workload under a 5-second burst from an 8-thread antagonist, which raises performance-tier latency to 1.5× that of the capacity tier (grey region).* 81

5.3 **Hecaton Balancing Paths.** *(a) Clean mirrored: remap only. (b) Dirty mirrored: overwrite the stale copy, then remap. (c) Single-copy into free space: write to an empty slot, then remap.* 85

5.4 **Read Set Identification.** *Three cases: (a) read-only, (b) $T_{write} \leq \tau_r$, (c) $T_{write} > \tau_r$ with exponential penalty.* 89

5.5 **Performance tier contention: read-only workload.** *A stream-like workload that uniformly randomly accesses a 32 GB read-only working set runs on the performance tier. At 60s, an 8-thread antagonist begins generating 30 GB/s of traffic (75% of performance tier bandwidth). Subplots show throughput (top), migration traffic (middle), and performance tier latency (bottom).* 95

5.6 **Performance tier contention: 95/5 read/write workload.** *Same antagonist setup as Figure 5.5, but the workload mixes 95% reads and 5% writes over a 32 GB working set with a 5% write set. Subplots show throughput (top), migration traffic (middle), and performance tier latency (bottom).* 96

5.7 **Sensitivity study: convergence vs. interference and read-set identification.** *Same experiment setup as Figure 5.5. x-axis: convergence time. (a) y-axis: performance-tier latency during convergence; sweeps Colloid-s's migration limit (1–10 GB/s). (b) y-axis: migration volume; Hecaton with vs. without read-set identification at 50% write set and 50% mirroring space.* 98

5.8 **Sensitivity study: migration volume.** *Same experiment setup as Figure 5.5. x-axis: convergence time; y-axis: migration volume during convergence. (a) varies working set (4–32 GB). (b) varies Hecaton's mirroring space (1.0×–0× WS). (c) varies write-set percentage (0–1.0).* . 99

5.9 **Load burst.** *A multichase workload with a 32 GB working set under a single 5-second burst of contention on the performance tier. The gray region marks the burst period.* 100

5.10 **Periodic bursts.** *The workload has a 16 GB resident working set. Each burst lasts 5 s and repeats every 10 s, with 16 antagonist threads active during bursts and 2 during idle periods. Gray regions mark burst periods.* 100

5.11 **Dynamic workload throughput.** *Steady-state throughput of the multichase workload under periodic bursts as Figure 5.10. (a) Varies burst intensity from 2/4 to 2/16 (low-load/high-load threads) with fixed burst frequency (5s burst / 5s idle). (b) Varies burst frequency from 5s burst / 2.5s idle to 5s burst / 15s idle with fixed intensity at 2/16. Across all settings, Hecaton migrates 19–21 GB in total, while Colloid-s-1, Colloid-s-10, and Colloid migrate up to 131, 540, and 138 GB, respectively.* 101

5.12 **Realistic workloads under periodic bursts.** *Same burst setup as Figure 5.10. (a) Steady-state throughput normalized to HeMem. (b) Total data migrated during the experiments normalized to Hecaton.* . . . 104

Abstract

Heterogeneous storage and memory systems combine multiple device types to balance performance, capacity, and cost, typically pairing a faster, smaller *performance* device with a slower, larger *capacity* device. For decades, these systems have been *hierarchical*: the performance device is orders of magnitude faster, so the capacity device's performance can be safely ignored. Modern devices break this assumption. The emergence of low-latency SSDs, persistent memory, and CXL memory has greatly narrowed the performance gap between neighboring devices, producing *non-hierarchical* storage and memory systems in which the capacity tier offers comparable bandwidth and, under contention, can even be faster than the performance tier. Both classic and state-of-the-art methods fall short on non-hierarchical architectures. For example, caching, which replicates hot data in the performance device, and tiering, which migrates hot data to it, both optimize for the performance tier; once that tier saturates, the capacity tier remains underutilized and overall performance is bounded by a single device. This dissertation addresses this challenge in three parts, each identifying the limitations of classic and state-of-the-art approaches in non-hierarchical storage and memory systems and proposing new designs that aim to dynamically utilize all devices optimally.

The first part revisits caching for modern storage hierarchies. We introduce *Non-Hierarchical Caching* (NHC), an approach that layers a lightweight, feedback-driven scheduler on top of an existing cache controller. The key insight is that a cache hit need not be served from the performance device: when that device saturates, NHC redirects a fraction of hits to the copy on the capacity device, exploiting its unused bandwidth. We implement NHC as

Orthus-CAS, a kernel module within the open-source Open CAS framework, with configurable optimization targets including throughput, average latency, and tail latency. Compared to classic caching, Orthus-CAS achieves up to 2× higher throughput when the performance device is saturated, and it substantially outperforms prior cache-bypass approaches under dynamic workloads.

The second part redesigns tiering, which avoids caching's duplicated capacity but keeps a single copy of each block and must migrate data to shift load. We introduce *Mirror-Optimized Storage Tiering* (MOST), a tier-based approach that mirrors a small amount of hot data across devices and balances load by routing requests among the copies instead of migrating data. A simple, self-adjusting optimization mechanism controls the routing ratio online, requiring no prior knowledge of the workload or devices. We implement MOST as Cerberus, a storage management layer within the widely used flash cache library CacheLib. Across two storage hierarchies, Cerberus achieves up to 2.34× higher throughput and reduces P99 latency by up to 75% compared to state-of-the-art systems on static workloads, improves throughput by up to 1.86× across four production cache workloads, and reduces device writes by up to 84% under dynamic workloads.

The third part extends load balancing to heterogeneous memory, where local DRAM is paired with second-level memory such as CXL devices and memory-interconnect contention can invert the latency relationship between tiers. Existing contention-aware systems rebalance by migrating pages, but migration adds traffic to the already contended memory subsystem, slows convergence, interferes with foreground requests, and repeats data movement under bursty load. We introduce Hecaton, a heterogeneous memory management system built on *mapping-based load balancing*: Hecaton maintains mirrored copies of read-mostly pages across tiers and balances load through page table remapping without data copying under contention. A lightweight latency-feedback controller, low-overhead reactive and proactive mirroring, and a write-recency test for selecting pages worth mirroring make this approach practical. Across microbenchmarks, bursty workloads, and

realistic applications, Hecaton outperforms contention-oblivious tiering by up to 2.60× and the best migration-based baseline by 1.64×, converges up to 32.8× faster than the lowest-interference migration variant, and migrates up to 34× less data.

Together, these three systems introduce new designs that make better use of all devices in non-hierarchical storage and memory systems.

Chapter 1

Introduction

> *"Ideally one would desire an indefinitely large memory capacity . . . It does not seem possible physically . . . We are therefore forced to recognize the possibility of constructing a* ***hierarchy*** *of memories,* ***each of which has greater capacity than the preceding but which is less quickly accessible****."*
>
> – Burks, Goldstine, and von Neumann, 1946 [27]

Heterogeneous systems have been central to computer system design since its inception [27]. A heterogeneous system refers to an architecture that combines multiple, distinct types of devices to balance performance, capacity, and cost. In storage systems, such heterogeneous architectures are commonplace. For example, DRAM has long served as a cache above hard drives (e.g., the Linux page cache [33]), and SSDs have been widely deployed as a caching layer above hard drives since they became commercially available [2]. More recently, advances in memory technology and growing demand for memory capacity and bandwidth have driven memory systems from a single-tier DRAM architecture to a heterogeneous one, pairing DRAM with newer technologies such as CXL memory and persistent memory to deliver higher capacity and bandwidth at lower cost.

To understand how these systems are managed, consider a simplified two-device system: a faster, smaller *performance* device and a slower, larger *capacity*

device. We refer to the position each device occupies in the hierarchy as the *performance tier* and the *capacity tier*, and we use these terms throughout this dissertation. For decades, such heterogeneous storage and memory systems have been managed with two classic approaches: caching [23, 89], which replicates frequently accessed data in the performance device, and tiering [64, 77, 96, 137], which periodically migrates hot data to the performance device. These classic methods all make an assumption that the performance device is orders of magnitude faster than the capacity device, so the capacity device's performance can be ignored. An example is DRAM caching above hard drives: HDD performance is so low that it can be safely ignored, so caching research has focused on replacement algorithms that maximize the cache hit rate [80, 89] for decades. We call such an architecture, in which the performance gap between tiers is large enough that the capacity tier can be safely ignored, a *hierarchical* architecture.

However, this assumption is breaking down as modern heterogeneous storage and memory systems evolve from a hierarchical architecture toward a *non-hierarchical* one, in which the performance gap between the two devices is small enough that the capacity device's performance can no longer be ignored. This evolution is driven by the emergence of new devices, including low-latency SSDs [8, 10], persistent memory [7], and CXL devices [35]. As we show in Chapter 2, these new devices have performance close to that of their neighboring tiers in the device pyramid. For example, a low-latency SSD is a block storage device with much lower latency than a traditional flash SSD. These devices offer significantly lower latency than traditional flash SSDs, but their bandwidth is comparable. Pairing a low-latency SSD with a traditional NAND SSD therefore yields a heterogeneous storage system that delivers ultra-low latency and large capacity for latency-sensitive workloads—and because the bandwidth of the two tiers is comparable, the traditional SSD's bandwidth cannot be ignored. We observe a similar phenomenon in heterogeneous memory systems, where local DRAM is paired with a second-level memory device such as CXL memory—a byte-addressable, cache-coherent device attached over PCIe that delivers large capacity with

DRAM-like access latency. CXL memory offers comparable bandwidth to DRAM and is only 2–3× slower in latency, and under contention it can even become faster than DRAM [117].

We call these two classes of systems *non-hierarchical storage* and *non-hierarchical memory*. A non-hierarchical storage system is a heterogeneous storage system whose tiers have close performance: for example, a low-latency SSD paired with a flash SSD, where the two tiers deliver comparable bandwidth. Similarly, a non-hierarchical memory system is a heterogeneous memory system whose tiers have close performance: for example, local DRAM paired with CXL memory, where the capacity tier matches the performance tier's bandwidth and trails its latency by only a small factor. In both classes, the capacity tier contributes a substantial fraction of the system's aggregate performance and can no longer be treated as a slow backing tier. We use these two terms throughout this dissertation to distinguish such systems from classic hierarchical architectures.

As we show in this dissertation, classic caching and tiering, both designed under the hierarchical assumption, fall short in non-hierarchical storage and memory systems. Classic caching [23, 89] keeps copies of hot data in the performance tier and serves all cache hits from there; it has no mechanism to redirect hits to the capacity tier. Once the performance tier saturates, the capacity tier remains underutilized while the majority of traffic continues to queue on the overloaded performance tier. Classic tiering [64, 77, 96] keeps only a single copy of each block and periodically migrates hot data onto the performance tier. Because the hot copy resides only in the performance tier and serves the majority of requests, classic tiering faces the same capacity-underutilization problem as classic caching. Non-hierarchical storage and memory systems thus raise a new challenge: systems can no longer optimize for the performance tier alone; they must utilize all tiers effectively.

This dissertation takes a step toward addressing this challenge. Specifically, this dissertation has three parts. First, we rethink how caching should be done on modern heterogeneous storage systems (Chapter 3). Second, we revisit tiering and show how to optimize it for modern heterogeneous

storage systems (Chapter 4). Third, we study how load balancing can be performed more efficiently and quickly on modern heterogeneous memory systems (Chapter 5). In each project, we identify how classic approaches and state-of-the-art systems fall short under non-hierarchical architectures, and propose new designs that address these shortcomings and efficiently utilize all devices in the heterogeneous system. In the following sections, we expand on each of the three parts, summarize the full list of contributions, and outline the remaining chapters of this dissertation.

1.1 Optimizing Caching for Modern Non-Hierarchical Storage

We start by revisiting caching, the dominant approach for managing storage hierarchies for decades. Classic caching is built on a simple principle: maximize the fraction of accesses served by the performance device. As a result, classic caching has focused on designing increasingly advanced replacement and promotion algorithms (e.g., LRU [89], 2Q [55], ARC [80], and FIFO [22]) that maximize the cache hit rate under different workloads. These designs work well in traditional hierarchies that pair devices with order-of-magnitude performance differences, such as DRAM and disk. In such hierarchies, the capacity device is so much slower than the performance device that its performance can be safely ignored.

Modern storage hierarchies, however, are non-hierarchical, as discussed at the start of this chapter: the capacity device's bandwidth approaches or even exceeds that of the performance device. Classic caching cannot exploit this. Under heavy load, it saturates the performance device while leaving the capacity device's bandwidth largely unused, so aggregate throughput is bounded by the performance device alone.

Prior work has addressed some of these limitations by offloading excess writes from SSDs to the underlying hard drives [20, 59]. As we show later in the evaluation, however, these approaches share two critical limits: they

do not redirect cache hits to the capacity device, and they do not adapt to changing workloads and concurrency levels, which is critical for modern devices.

We introduce *Non-Hierarchical Caching* (NHC), a caching approach that layers a lightweight, feedback-driven scheduler on top of an existing cache controller. The key idea behind NHC is simple but efficient. When a request hits the cache, it does not always need to be served from the copy on the performance tier; it can also be served from the copy on the capacity tier. When the performance device saturates, NHC therefore redirects a fraction of clean cache hits to the capacity device, exploiting its unused bandwidth and reducing load on the performance device.

To achieve this, NHC deploys a scheduler that continuously monitors the end-to-end performance of both devices and adjusts a load admit ratio that determines what fraction of read hits are redirected to the capacity device. Under light load, NHC behaves like classic caching and serves all hits from the performance device; once the performance device saturates, NHC offloads excess hits to the capacity device to exploit its bandwidth.

This work was done jointly with colleagues. My primary contributions were the implementation of NHC as Orthus-CAS (a kernel module within the open-source Open CAS framework [6]), the implementation of configurable target metrics (average latency, and P99 tail latency) on top of NHC's scheduler, the re-implementation of two prior cache-bypass systems within the same framework as baselines, and the experimental comparison between Orthus-CAS and these baselines. Compared to classic caching, Orthus-CAS achieves up to 2× higher throughput by exploiting the capacity device's bandwidth when the performance device is saturated. Compared to previous state-of-the-art approaches, Orthus-CAS substantially outperforms all baselines under dynamic workloads. This work is based on the paper *The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus* published in FAST 2021 [123].

1.2 Getting the MOST with Mirror-Optimized Storage Tiering

While NHC addresses the challenge of read load balancing in a caching model, it has two major drawbacks. The first is that NHC cannot balance writes and performs poorly under write-intensive workloads. The second is more fundamental: a limitation of the caching model itself—the performance tier is used solely for duplication, wasting its capacity. The second part of this dissertation addresses these two limitations through an architectural redesign: we explore multiple classic designs and propose a new hybrid architecture that achieves efficient load balancing under the tiering model with a small amount of mirroring.

Storage hierarchies have been essential to computer system design since its inception [27]. In persistent storage systems, such hierarchies are commonplace; for example, SSDs are commonly used as a caching layer above hard drives [86]; AutoRAID uses a mirrored set of disks for performance on top of a log-structured RAID-5 layout for capacity [120].

However, as recent work has shown [50, 123], new device types pose challenges in the construction of storage hierarchies. Specifically, new non-volatile memories [7] and low-latency SSDs [8, 10] have performance and capacity profiles that overlap. As such, arranging devices into a strict hierarchy is difficult [50, 123]; doing so ensures that the peak potential of the storage system is not realized.

We are thus left with a significant challenge: how should we manage a collection of modern devices so as to maximize performance while minimizing space overheads? As we discuss in the second part of this dissertation, classic techniques fall short.

To understand the issues better, consider a simplified two-device system, with a faster/smaller "performance" device and a slower/larger "capacity" device. The first family of approaches we call "single copy", as they keep only a single copy of each data block in storage. One well-known single-copy approach is tiering [18, 37, 46, 63, 64, 77, 94, 96, 105, 108, 116, 121, 128,

131, 137, 139], which (slowly) migrates blocks between tiers based on the hotness or some other optimization goal. Tiering is space-efficient, as there is one copy of each block; however, a tiering-based system cannot efficiently adapt to changes in load because data needs to be migrated around the tiers. Striping [30, 101] is another single-copy approach that statically allocates blocks across devices but struggles to manage heterogeneous hierarchies or changes in workload [34].

The second family of approaches we call "multiple copy", as they replicate some (or all) blocks. A well-known multiple-copy approach is caching [6, 23, 40, 79, 88, 89, 93, 123, 129], which makes copies of popular blocks in the performance device. As a result, caching wastes capacity (as all copy-based approaches do); in addition, caching does not utilize the bandwidth of the capacity layer, thus falling short of peak performance. Mirroring [25, 30], a technique that replicates data across two devices, effectively balances the load for read requests by directing them to either device, yet also incurs a significant capacity cost.

The state-of-the-art approaches for handling heterogeneous memory and storage devices focus on tiering [32, 116]; at a high level, these previous works attempt to maximize the total throughput delivered by all devices by dynamically placing and migrating data across devices such that accesses to each device are proportional to their bandwidth or equalize access latency. In the second part of this dissertation, we introduce *Mirror-Optimized Storage Tiering* (MOST) and show that mirroring a small amount of hot data across devices, in combination with tiering, greatly improves performance and reduces device writes. Given these mirrored data, the host can dynamically route requests to different devices to effectively balance load instead of performing costly data migrations across devices like other approaches.

We find that adding a small amount of mirroring is advantageous to pure tiering for three primary reasons. First, mirrors enable the system to quickly react to workload changes by adjusting routing; pure tiering, on the other hand, must slowly migrate a significant amount of data to adjust load distribution. Second, mirrors reduce overall device writes

when handling dynamic workloads; migration-based approaches require extensive device writes to move data across tiers. Third, mirroring is more robust to fluctuations in device performance and prevents overreacting with unnecessary migrations. This effect is particularly pronounced for storage devices that suffer from poor tail latency or write-heavy workloads that frequently trigger background activity inside the SSD.

MOST is a novel approach optimized for the modern storage hierarchy. MOST mirrors a small amount of hot data within the classic tiering system, enabling flexible load distribution control. Under low load, MOST functions similarly to a classic tiering system, routing requests for mirrored data to the performance device to maximize its utilization. Under high load, MOST dynamically routes a portion of requests to the capacity copy of the mirrored data, thereby utilizing the bandwidth of the capacity device more effectively. Key to the success of MOST are the algorithms that manage its behavior. MOST is based on a simple, robust, and self-adjusting optimization mechanism that requires no prior workload knowledge and is capable of handling the varied performance characteristics of different storage hierarchies.

We implement MOST as a storage management layer named Cerberus within the widely-used flash cache library, CacheLib [2, 23, 79]. Our experiments are conducted on two storage hierarchies using three different real device types [5, 8, 13–15]. For static cache workloads, MOST achieves up to 2.34× higher throughput and reduces P99 latency by up to 75% compared to state-of-the-art systems (Colloid [116], HeMem [96], and Orthus [123]). Across four production cache workloads, MOST increases throughput by up to 1.86× and reduces P99 GET latency by up to 90% compared to these systems. Overall, Cerberus enhances throughput under intensive workloads and reduces writes by up to 84% compared to Colloid under dynamic workloads through its judicious use of mirroring across storage tiers. This work is based on the paper *Getting the MOST out of your Storage Hierarchy with Mirror-Optimized Storage Tiering* published in FAST 2026 [114].

1.3 Mapping-Based Balancing in Non-Hierarchical Memory

The first two parts of this dissertation address non-hierarchical storage systems by rethinking caching and tiering. The same phenomenon, however, arises in non-hierarchical memory systems, where local DRAM is paired with second-level memory such as CXL-attached memory: under interconnect contention, the loaded latency of the performance tier can rise above that of the capacity tier. The third part of this dissertation focuses on load balancing in non-hierarchical memory systems. We demonstrate the limitations of existing state-of-the-art memory tiering systems and propose an alternative approach that achieves faster and more efficient load balancing.

Driven by the growing capacity and bandwidth demands of memory-intensive workloads, heterogeneous memory has received significant attention and has been widely deployed in data centers [77, 138]. By integrating multiple memory technologies into a single system, it scales memory capacity and bandwidth beyond what DIMM-attached DRAM alone can provide [35, 54]. In such systems, a small, fast local DRAM serves as the performance tier, while a larger, slower memory device serves as the capacity tier, such as a CXL device [85, 103, 106, 133]. Classic tiered memory systems focus on efficiently utilizing the performance tier so that workloads benefit from the speed of DRAM while exploiting the larger capacity of the second tier [18, 56, 58, 64, 68, 77, 96, 98, 107, 126–128, 138].

These classic systems are built on an implicit assumption: the access latency of the performance tier is lower than that of the capacity tier, so hot pages should be retained in the performance tier as much as possible. For example, Nomad [126] mitigates thrashing by retaining a hot page's capacity-tier copy during and after its promotion to the performance tier. However, recent work has shown that this assumption breaks under memory interconnect contention [54, 116, 117]: the loaded latency of the performance tier can rise to several times that of the capacity tier, even when bandwidth is far from saturated. We call this *latency inversion*. In this regime, a contention-

oblivious placement that maximizes performance-tier utilization becomes suboptimal: since the capacity tier is now faster, shifting load to the capacity tier can improve overall performance [32, 116].

State-of-the-art systems such as BATMAN [32] and Colloid [116] address this problem by dynamically migrating pages across tiers, either to maximize aggregate bandwidth or to equalize the loaded access latency of the two tiers. However, these systems perform poorly under bursty contention, which is common in real workloads [41]: they adapt too slowly, interfere with the foreground workload, and incur excessive data copying. The key problem is that data copying sits on the load-balancing critical path under contention: copying becomes slow under the increased memory latency and disrupts the foreground workload by competing for bandwidth. This raises a question: how can we achieve fast and efficient load balancing in tiered memory while minimizing migration?

In the last part of this dissertation, we introduce *mapping-based load balancing*, a fast and low-cost alternative to migration-based load balancing for tiered memory. When the system is not contended, mapping-based load balancing creates mirrored copies of read-mostly pages across tiers; under contention, it balances load by updating page-table mappings without copying data.

Mapping-based load balancing is advantageous over migration-based load balancing for three primary reasons. *Fast convergence.* Mapping-based load balancing reacts to load changes only by remapping page-table entries; a migration-based approach must additionally copy data across tiers besides page-table remapping, which is slow under a contended memory interconnect. *Minimal foreground interference.* Remapping adds no traffic to the contended tier; in contrast, migration must inject additional traffic onto the already-contended tier, which increases access latency and interferes with the foreground workload. *No repeated data movement.* Mirrored copies persist across load shifts, so pages remapped to the capacity tier during a load burst can be remapped back when the burst ends, while migration must move the same data back and forth each time the load changes.

We build Hecaton, a tiered memory system that implements mapping-based load balancing on top of Colloid, the state-of-the-art contention-aware memory system, addressing three challenges. First, mapping-based load balancing can balance a large amount of load in a short time, so the controller algorithm itself can become the bottleneck; Hecaton addresses this with a simple but fast feedback controller that reacts to latency signals quickly. Second, mapping-based load balancing requires extra memory bandwidth to create mirrors; Hecaton hides this overhead with reactive mirroring, which creates mirrors from existing migrations at no extra cost, and proactive mirroring, which creates mirrors when the system has spare resources. Third, mirrors of frequently written pages are quickly invalidated; Hecaton uses a read-set identification algorithm with a write-recency test to find the right pages to mirror.

We evaluate Hecaton against HeMem [96] (classic tiering, no balancing) and three Colloid configurations (migration-based balancing) on dynamic workloads, bursty workloads, and realistic memory-intensive applications [21, 39, 64, 81, 110, 113, 115]. Across all the experiments, Hecaton achieves up to 2.60× HeMem's performance when the performance tier is contended. Under contention, Hecaton converges up to 32.8× faster than the lowest-interference Colloid variant. Across bursty workloads and realistic memory-intensive applications, Hecaton achieves up to 1.64× the throughput of the best-performing Colloid variant. Compared to Colloid with a dynamic migration limit, Hecaton migrates up to 34× less data.

1.4 Contributions and Highlights

We list the main contributions of this dissertation, organized by project.

Optimizing Caching for Modern Non-Hierarchical Storage.

- We propose *Non-Hierarchical Caching* (NHC), which augments any classic cache controller with a lightweight, feedback-driven scheduler

that redirects a fraction of read hits to the capacity device under saturation, requiring no prior workload or device knowledge.

- We implement NHC as Orthus-CAS in the open-source Open CAS framework [6], and implement two previous state-of-the-art systems—SIB [59] and LBICA [20]—as baselines within the same framework for comparison.
- We evaluate NHC against state-of-the-art baselines across both static and dynamic workloads. Orthus-CAS achieves up to 2× higher throughput than classic caching and substantially outperforms state-of-the-art baselines under dynamic workloads.

Getting the MOST with Mirror-Optimized Storage Tiering.

- We classify classic storage-management approaches into single-copy (tiering, striping) and multiple-copy (caching, mirroring) families, and show that no classic approach offers tiering's space efficiency together with mirroring's load-balancing flexibility.
- We design Mirror-Optimized Storage Tiering (MOST), a hybrid approach that mirrors only a small subset of hot data within a classic tiering system. Under low load, MOST behaves like classic tiering; under high load, it routes requests across mirrored copies to balance load through routing rather than migration. A simple feedback-driven optimizer adjusts the routing ratio online.
- We implement MOST as Cerberus within the widely used flash cache library CacheLib [2, 23, 79], and evaluate it on two storage hierarchies. Cerberus achieves up to 2.34× higher throughput, up to 75% lower P99 latency, and up to 84% fewer device writes than the state of the art (Colloid [116], HeMem [96], and Orthus [123]).

Mapping-Based Balancing in Non-Hierarchical Memory.

- We classify tiered memory systems into two classes: *contention-oblivious* systems (e.g., HeMem [96]) that perform poorly under interconnect contention, and *contention-aware* systems (e.g., BATMAN [32], Colloid [116]) that handle contention through migration. We show that contention-oblivious systems leave the capacity tier unused once the performance tier is contended, and that state-of-the-art contention-aware systems cannot adapt to bursty workloads without significantly interfering with the foreground workload.

- We introduce *mapping-based load balancing*, an alternative that maintains mirrored copies across tiers and rebalances load by updating page-table mappings, requiring no data movement on the contended interconnect. Mirrored copies persist across load shifts, eliminating the repeated data movement that migration-based systems incur.

- We design and build Hecaton, a tiered memory system that implements mapping-based load balancing on top of Colloid [116], addressing three challenges: a fast feedback controller that reacts to latency signals on millisecond timescales; reactive and proactive mirroring that hide the bandwidth cost of creating mirrors; and a read-set identification algorithm with a write-recency test that selects the right pages to mirror.

- We evaluate Hecaton against HeMem [96] (classic tiering, no balancing) and three Colloid configurations (migration-based balancing) on dynamic workloads, bursty workloads, and realistic memory-intensive applications. Hecaton achieves up to 2.60× HeMem's throughput when the performance tier is contended, converges up to 32.8× faster than the lowest-interference Colloid variant, and migrates up to 34× less data than Colloid.

1.5 Outline of Chapters

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

- **Heterogeneous Storage and Memory Systems.** Chapter 2 introduces the new device classes driving the shift toward non-hierarchical systems—CXL memory, persistent memory, and low-latency SSDs—and shows how each narrows the performance gap between neighboring tiers, breaking the traditional hierarchical assumption.

- **Optimizing Caching for Modern Non-Hierarchical Storage.** Chapter 3 presents Non-Hierarchical Caching (NHC) and its implementation in Orthus-CAS. We first show that the classic caching principle of maximizing hit rate breaks down once the performance device saturates. We then describe NHC's lightweight, feedback-driven scheduler, its implementation as a kernel module within the Open CAS framework, and an experimental comparison against prior cache-bypass schemes.

- **Getting the MOST with Mirror-Optimized Storage Tiering.** Chapter 4 presents Mirror-Optimized Storage Tiering (MOST) and its implementation in Cerberus. We first show that pure migration-based tiering converges too slowly to track dynamic workloads and incurs a significant amount of write traffic. We then describe MOST's hybrid design, in which only a small subset of hot data is mirrored, and evaluate Cerberus on two storage hierarchies across static and dynamic workloads.

- **Mapping-Based Balancing in Non-Hierarchical Memory.** Chapter 5 presents Hecaton, a tiered memory system that introduces mapping-based load balancing on top of Colloid. We first show that under memory interconnect contention, migration-based balancing cannot adapt to bursty workloads without significant interference with the foreground workload. We then describe Hecaton's design—its fast feedback controller, reactive and proactive mirroring, and read-set selection algorithm—and evaluate it across dynamic, bursty, and realistic memory-intensive workloads.

- **Related Work.** Chapter 6 discusses prior work on caching, tiering, and load balancing across both storage and memory hierarchies, and

positions the three projects of this dissertation within that landscape.

- **Conclusion and Future Work.** Chapter 7 summarizes the dissertation, outlines the future directions our work suggests, and reflects on the lessons learned.

Chapter 2

Heterogeneous Storage and Memory Systems

The modern storage and memory hierarchy has evolved significantly with the emergence of new device classes over the past decade, as illustrated in Figure 2.1. As we describe in this chapter, these new devices have narrowed the performance gap between neighboring tiers to within a small constant factor. Table 2.1 summarizes the latency, maximum bandwidth, and capacity of representative devices across the modern hierarchy. This chapter introduces the three device classes driving this change—CXL devices (§2.1), persistent memory (§2.2), and low-latency SSDs (§2.3)—and provides the background shared across the three research chapters of this dissertation.

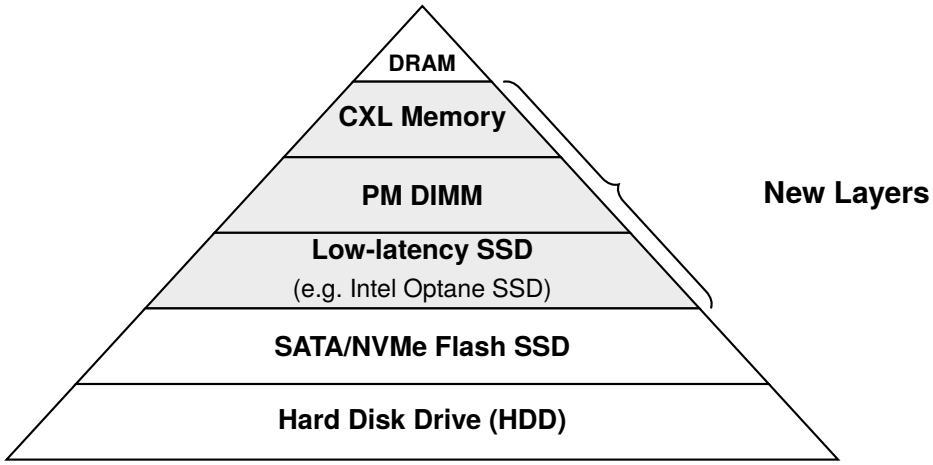


Figure 2.1: Modern Heterogeneous Storage and Memory Architecture. *Three new device classes—CXL memory, persistent memory (PM) DIMM, and low-latency SSD—bridge the performance gap between DRAM and SATA/NVMe flash SSD, narrowing the gap between neighboring tiers in the modern hierarchy.*

2.1 CXL Devices

The Compute Express Link (CXL) is an open industry standard that defines a family of interconnect protocols between CPUs and devices over the PCIe physical layer. CXL 1.0/1.1 introduces three protocols: *CXL.io* defines the basic semantics for device discovery, configuration, and non-coherent DMA, and is mandatory on every CXL device; *CXL.cache* lets a device coherently cache host memory; and *CXL.mem* lets the host access device-attached memory as cacheable memory in its own physical address space, so reads and writes to the device's memory can be cached in the CPU's caches exactly as accesses to local DRAM are.

Based on these protocols, the CXL 1.0/1.1 specification defines three device types that capture different usage models. *Type 1* devices implement CXL.io and CXL.cache; they are accelerators such as smart NICs that cache host memory but expose no local memory to the host. *Type 2* devices implement all three protocols; they are accelerators such as GPUs and FPGAs whose local memory is mapped into the host's address space. *Type 3* devices implement CXL.io and CXL.mem and are used for memory bandwidth and

Device type	Interface	Latency	Max. Bandwidth	Capacity
DRAM	DDR5	123 ns	64 GB/s	16–128 GB
CXL memory	CXL.mem	239 ns	52 GB/s	768 GB
PM	DDR	305 ns	8.10 GB/s	128–512 GB
CXL SSD	CXL.mem	0.6 – 10 μs	4.5 GB/s	1–2 TB
Low-latency SSD	PCIe 4.0 NVMe	3.5 μs	7.2 GB/s	0.4–3.2 TB
Flash SSD	PCIe 5.0 NVMe	60 μs	14.0 GB/s	3.2–30.72 TB
	PCIe 4.0 NVMe	45.5 μs	7.45 GB/s	1–4 TB
	PCIe 3.0 NVMe	86 μs	3.5 GB/s	3.2–15.36 TB
SATA SSD	SATA	120 μs	560 MB/s	0.48–7.68 TB
Hard disk	SATA	4.16 ms	269 MB/s	30–32 TB

Table 2.1: Performance of Representative Devices. *DRAM, Micron DDR5-8000 RDIMM [83]; CXL memory, CXL-D Type 3 DRAM memory expander (CXL 1.1, PCIe 5.0 ×16, 4×DDR5); Persistent memory, Intel Optane PMem 200 [51]; CXL.mem + SSD, Samsung CMM-H prototype; Low-latency SSD, Intel Optane SSD P5800X [52]; Flash SSD (PCIe 3.0), Micron 9300 PRO/MAX [82]; Flash SSD (PCIe 4.0), Samsung 990 PRO [102]; Flash SSD (PCIe 5.0), Micron 9550 PRO/MAX [84]; SATA SSD, Samsung PM893 [104]; Hard disk, WD Ultrastar DC HC690 (7200 RPM) [119]. Throughput is per device/module unless otherwise stated.*

capacity expansion. Because they attach over PCIe rather than the DDR bus, they scale memory beyond the limited number of DDR channels a CPU can support, at lower cost and power. This dissertation focuses on Type 3 devices, which connect to the host through PCIe and expose their memory through CXL.mem as a CPU-less NUMA node. CXL Type 3 devices are now commercially available [85, 103, 106] and deliver latency within an order of magnitude of local DRAM (2–3× higher [67]), making them attractive as a second-tier memory alongside DRAM. A more recent development is the CXL-SSD [133, 134], a research-prototype Type 3 device that combines a small DRAM cache (e.g., 48 GB) with a NAND flash backend. It exposes a

byte-addressable, cache-coherent memory interface, but its access latency is hybrid—memory-class on a DRAM-cache hit and Flash-class on a miss.

Recent progress in CXL has attracted significant attention from industry [77] and academia [17, 64, 68, 116] as a response to the *memory wall* [43] challenge: DRAM capacity and bandwidth cannot scale to keep up with the growing memory demands of memory-intensive workloads. This advances the single-DRAM model into a multi-tier heterogeneous memory architecture, in which devices with memory-like semantics deliver cost-efficient scaling of memory bandwidth and capacity. Efficiently managing such an architecture, however, becomes a new challenge. Table 2.1 summarizes the performance characteristics of representative commercial CXL devices. The key observation is that they deliver performance comparable to DRAM, so pairing CXL memory with local DRAM yields a non-hierarchical memory system, on which classic approaches that optimize only for the performance tier fall short. Among the challenges that arise in a non-hierarchical memory system, the system's ability to balance load across the two memory tiers is one of the most critical to realizing the full performance of all devices. This problem does not exist in classic hierarchical systems, where the capacity tier's performance is low enough to be ignored. The last part of this dissertation addresses this gap with load-balancing techniques for heterogeneous memory systems.

2.2 Persistent Memory

Persistent memory (PM) refers to a class of technologies that provide byte-addressable storage with two defining characteristics: memory-like access through ordinary load and store instructions, and persistence of stored data across power loss. Table 2.1 reports the performance of Intel Optane DC PM, the first commercial PM product, based on 3D XPoint. Because of these characteristics, PM has been explored along two distinct paths. First, as a cheaper alternative to DRAM, where its persistence is ignored and it is used solely as a slow memory tier [96]. In this role, PM serves as the second

tier of a heterogeneous memory system: as shown in Table 2.1, its latency is roughly 2.5× that of DRAM, and its bandwidth, though lower, remains substantial. Second, PM can be paired with flash SSD as the performance tier of a heterogeneous storage system, exploiting its low latency and byte-addressable persistence. Compared with flash SSD, PM offers significantly lower latency but bandwidth that is comparable to, and in some cases lower than, high-end Gen 5 flash SSDs. In both cases—whether PM serves as the performance tier in storage or the capacity tier in memory—the two tiers deliver comparable performance, and neither can be ignored. Therefore, in a PM-involved heterogeneous architecture, how to efficiently use both devices becomes the critical challenge. Although the commercial release of PM drew significant attention from both academia and industry, Intel discontinued the Optane product line in 2022. Nevertheless, we believe our designs for non-hierarchical storage and memory systems remain applicable and generalize to future byte-addressable persistent devices.

2.3 Low-latency SSDs

Low-latency SSDs are a class of block storage devices that deliver latency below 10 μs, an order of magnitude lower than the ~80 μs typical of traditional flash SSDs based on TLC or QLC NAND. Commercial devices are built either on persistent memory technology (3D XPoint [8]) or on latency-optimized SLC NAND (XL-FLASH [10]). Their capacity, however, remains modest relative to mainstream flash. Pairing them with capacity-efficient flash SSDs yields a heterogeneous storage architecture that combines low latency with large capacity. Table 2.1 reports the performance of Intel Optane SSD P5800X, the first commercial low-latency SSD. Its latency is far below that of traditional flash SSDs, but its bandwidth is comparable when both use the same PCIe interface. As a result, pairing it with a flash SSD as the capacity tier produces a non-hierarchical storage system, with the same characteristics we observe in CXL memory and persistent memory.

2.4 Conclusion

For decades, heterogeneous systems have been managed around a single goal: maximizing utilization of the performance tier. However, as shown in this chapter, this assumption no longer holds: the new device classes have narrowed the performance gap between neighboring tiers to within a small constant factor, turning modern storage and memory systems into non-hierarchical ones. The classic problem of utilizing the performance tier remains, but the capacity tier can no longer be ignored. The three parts of this dissertation examine this problem and rethink how non-hierarchical storage and memory systems should be managed.

Chapter 3

Optimizing Caching for Modern Non-Hierarchical Storage

In this chapter, we examine how the classic caching principle—maximize hit rate, serve all hits from the performance device—falls short on non-hierarchical storage systems, and introduce *Non-Hierarchical Caching (NHC)*, a novel caching approach that dynamically offloads excess traffic from the performance device to the capacity device when doing so improves the chosen performance metric.

This work was done jointly with colleagues; my primary contributions were (i) implementing NHC as Orthus-CAS on top of the Open CAS framework, (ii) extending NHC's scheduler to support latency-based target metrics (average latency, and P99 tail latency), (iii) implementing SIB and LBICA as baselines within the same framework, and (iv) running the experimental comparison presented in this chapter. The rest of this chapter is organized as follows. §3.1 provides background and presents the design of NHC. §3.2 describes our implementation of Orthus-CAS and the baselines within the Open CAS framework. §3.3 evaluates Orthus-CAS against the baselines under static and dynamic workloads and across different performance targets. §3.4 concludes.

3.1 Background

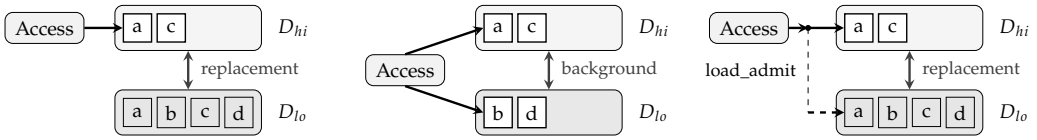
We first motivate why classic caching falls short on non-hierarchical storage systems (§3.1.1). We then present the design of Non-Hierarchical Caching (NHC, §3.1.2) and describe the two previous state-of-the-art systems, SIB and LBICA, that we compare against (§3.1.3). The Open CAS framework on which Orthus-CAS and our baselines are built is introduced in the implementation section (§3.2.1).

3.1.1 Motivation

A storage hierarchy combines multiple heterogeneous devices with policies for moving data among them. For simplicity, we assume a two-device hierarchy with a performance device D_{hi} and a capacity device D_{lo}: D_{hi} is typically smaller, faster, and more expensive per byte, while D_{lo} is larger, slower, and cheaper. Two classic approaches are used to manage such a hierarchy: *caching* and *tiering* (Figure 3.1).

Caching and tiering. With caching, popular (hot) data is copied from D_{lo} into D_{hi} on each miss; the cache evicts less popular (cold) data using a replacement policy such as LRU, ARC, or LFU. The granularity of data movement is small, usually 4-KB blocks, and the cache reacts quickly to workload changes. Tiering also keeps hot data in D_{hi} but, unlike caching, does not necessarily promote on misses: data is served directly from whichever tier it lives on, and migrations are performed periodically on longer time scales (hours or days). Both approaches share the same assumption: maximize the fraction of accesses served from D_{hi}.

Storage devices are no longer hierarchical. Storage systems are undergoing a rapid transformation. Persistent memory, low-latency SSDs, NVMe Flash SSDs, and SATA Flash SSDs now span a range of latency, bandwidth, and cost, but their performance is not strictly ordered. Table 3.1 lists representative device characteristics. Latency is strictly ordered, but bandwidth differences between adjacent rows are within a small factor, and write performance can



(a) Caching **(b) Tiering** **(c) Non-Hierarchical Caching**

Figure 3.1: Caching, Tiering, and Non-Hierarchical Caching. *(a) Caching copies hot data to the performance device D_{hi} on miss and serves all hits from D_{hi}, with replacement traffic flowing $D_{hi} \rightarrow D_{lo}$. (b) Tiering splits access between D_{hi} and D_{lo} and migrates data between them in the background on long time scales. (c) Non-hierarchical caching keeps D_{hi} populated like classic caching but, when D_{hi} saturates, redirects a tunable fraction of hits to D_{lo} via the* `load_admit` *switch (dashed).*

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

Table 3.1: Diversified Storage Devices. *Representative latency, bandwidth, and cost for modern storage and memory devices. Data taken from SK Hynix DRAM (DDR4, 16 GB), Intel Optane DCPM, low-latency SSDs (Intel Optane SSD 905P, Micron X100), NVMe Flash SSD (Samsung 970 Pro), and SATA Flash SSD (Intel 520 SSD); low-latency and NVMe Flash SSDs assume PCIe 3.0. Bandwidth and latency differences between adjacent rows are now within an order of magnitude, breaking the assumption that the hierarchy is strict.*

even invert under concurrency: a heavily loaded NVDIMM, for example, can deliver less write throughput than a low-latency SSD. The performance of new devices also depends on the workload, including read/write ratio, level of concurrency, and access pattern. As a result, a placement that maximizes use of D_{hi} no longer maximizes overall throughput on these hierarchies: when D_{hi} saturates, its loaded performance may be no better, or even worse, than D_{lo}, and D_{lo}'s bandwidth is left unused.

Implication. We make two key observations from Table 3.1. First, the storage

hierarchy is no longer a hierarchy in the traditional sense: adjacent layers can have comparable or overlapping performance. Second, caching and tiering policies that target only D_{hi} utilization leave aggregate bandwidth unused. A different approach is needed: one that delivers classic caching's behavior under light load but exploits D_{lo}'s bandwidth when D_{hi} saturates, without requiring static knowledge of device characteristics or workload parameters. This motivates Non-Hierarchical Caching.

3.1.2 Non-Hierarchical Caching

We now present the design of Non-Hierarchical Caching (NHC). NHC's central mechanism is simple: when the performance device is saturated, the system probabilistically redirects a fraction of read hits to the capacity device, choosing that fraction online from end-to-end performance feedback. The rest of this subsection formalizes NHC's setting (§3.1.2.1), enumerates its design goals (§3.1.2.2), describes its architecture (§3.1.2.3), and presents the scheduler algorithm (§3.1.2.4).

3.1.2.1 Formal Definitions

We assume a two-device hierarchy with D_{hi} and D_{lo}. Caching performance depends on how the workload is distributed across these devices. Let W denote the total workload over a time period dt, modeled as a finite set of accesses to data items. Let $w \subseteq W$ be the subset served by D_{hi}, with the complement $W \setminus w$ served by D_{lo}. We write total performance over the period as

$$P(W, w) \;=\; p_{hi}(w) + p_{lo}(W \setminus w),$$

where p_{hi} and p_{lo} are the per-device performance functions. We make three assumptions:

Assumption 1: Devices have a performance upper bound. Each device has a maximum delivered performance for the current workload, denoted L_{hi} and L_{lo}. We write w_0 for the smallest subset of w such that $p_{hi}(w_0) = L_{hi}$.

Assumption 2: More workload does not reduce performance. p_{hi} and p_{lo} are monotonically non-decreasing in the size of their input. (HDD performance can drop under random load, but this assumption holds for high-performance devices such as DRAM, NVM, and SSDs.)

Assumption 3: Before reaching the upper bound, p_{hi} has a larger gradient than p_{lo}. Adding workload to D_{hi} yields more performance per unit work than adding the same to D_{lo}, until D_{hi} saturates.

Classic caching is then an approach that maximizes $p_{hi}(w)$ alone, i.e., it searches for the working set w_{max} that maximizes $p_{hi}(w)$ (equivalently, D_{hi}'s hit rate under standard cache models). NHC's key insight is that whenever $w_0 < w_{max}$ (i.e., the working set on D_{hi} is larger than what is needed to saturate it), the fraction $w_{max} - w_0$ can be redirected to D_{lo} without reducing p_{hi}, and D_{lo} can deliver additional performance for those redirected accesses. This is why NHC can always match or exceed classic caching.

3.1.2.2 Design Goals

NHC is guided by three design goals.

Perform as well as or better than classic caching. Classic caching optimizes for hit rate by finding the best data to place on the performance device; this strategy maximizes hit rate but degrades throughput once the performance device saturates. NHC must match classic caching under light load (where the hit-rate strategy is correct) and exceed it under heavy load by exploiting the capacity device's bandwidth.

Require no special knowledge or configuration. NHC should make no assumptions about device characteristics (e.g., bandwidth ratios, latency profiles) or workload parameters (e.g., read/write ratios, working set size). Because modern devices have workload-dependent performance, any approach based on preconfigured thresholds becomes stale as conditions change. NHC should instead learn the right behavior online from end-to-end performance feedback.

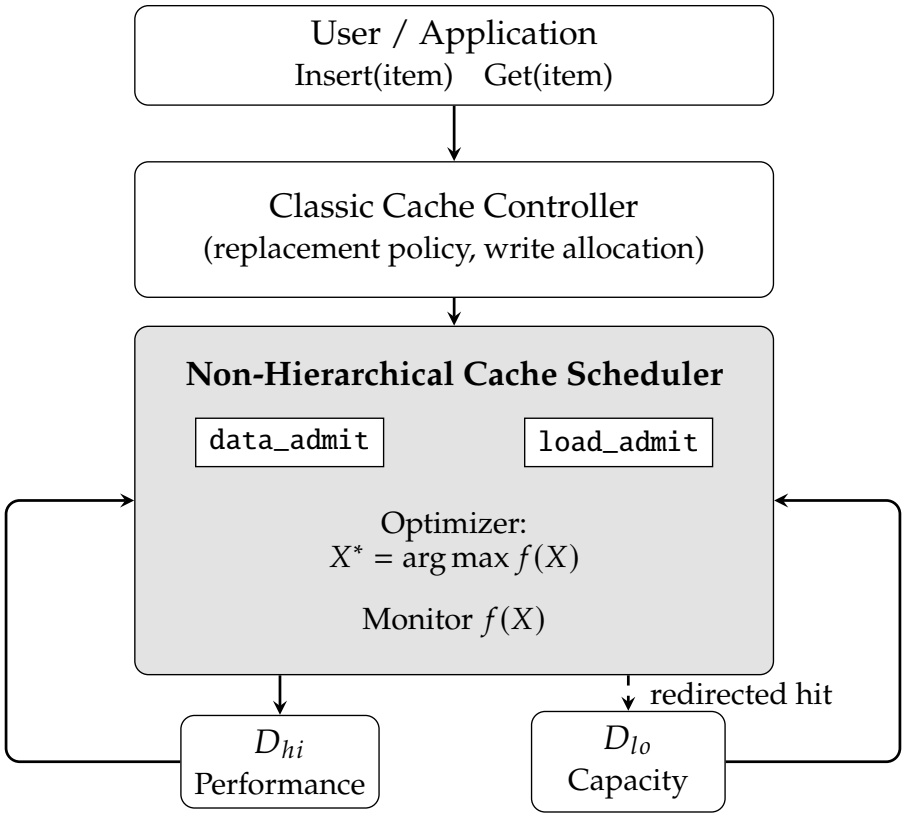


Figure 3.2: Non-Hierarchical Caching Architecture. *NHC adds two decision points to a classic cache controller:* `data_admit` *controls whether a read miss promotes a block to D_{hi}, and* `load_admit` *controls the fraction of read hits served from D_{hi} versus redirected to D_{lo}. The scheduler monitors a target metric $f(X)$ and tunes both knobs in feedback. NHC is transparent to users; any classic caching implementation can be upgraded to be an NHC one.*

Be robust to dynamic workloads. Workloads vary: load changes over the day, the hit rate shifts as working sets move, and read/write ratios fluctuate. NHC must adapt quickly to these changes.

3.1.2.3 Architecture

At a high level, NHC layers an *NHC scheduler* on top of an existing classic cache controller (e.g., Open CAS), as shown in Figure 3.2. The classic controller manages which data blocks reside on the performance device—

its replacement policy (LRU, ARC, etc.) remains unchanged. The NHC scheduler sits above it and decides two things:

- `data_admit`: whether the classic cache controller is permitted to admit and replace data on the performance device at all.
- `load_admit`: a scalar in $[0, 1]$ giving the fraction of clean read hits to redirect from the performance device to the capacity device.

When `load_admit`=0, every hit is served from the performance device (classic caching). When `load_admit`=r, an incoming clean read hit is routed to the capacity device with probability r and to the performance device with probability $1 - r$. The scheduler's job is to find the value of `load_admit` that maximizes a user-specified target metric (e.g. throughput, average latency, or tail latency) without requiring any knowledge of the underlying devices.

3.1.2.4 Cache Scheduler Algorithm

The NHC scheduler operates in two states that alternate depending on workload conditions, summarized in Algorithm 1.

State 1: Improve hit rate. The scheduler lets the classic cache controller warm up. It sets `data_admit`=TRUE (so the controller can promote new data) and `load_admit`=0. The scheduler monitors the hit rate and remains in this state until the hit rate has stabilized. Intuitively, there is no point offloading hits to the capacity device while the cache itself is still not warmed up. Therefore, NHC first lets the controller find the right working set and then balances load if necessary.

State 2: Adjust load between devices. Once the hit rate has stabilized, the scheduler freezes the cache admission (`data_admit`=FALSE) to prevent promotion/eviction traffic from interfering with load-balancing decisions, and begins tuning `load_admit`. It uses a simple gradient-based search: at each interval, it increases and decreases `load_admit` by a small step, measures the change in the target metric, and moves in the direction that gives the best performance. If the current value is the optimal, it holds steady. If the hit

Algorithm 1: Non-Hierarchical Cache Scheduler

Inputs: `cache`: classic cache controller; `step`: adjustment step for `load_admit`; $f(x)$: function that measures the target metric when `load_admit` = x over one interval

```
1  while true do
       // State 1:  Improve hit rate
2      data_admit ← true; load_admit ← 0.0
3      while cache.hit_rate is not stable do
4          sleep_a_while()
5      end
6      data_admit ← false; start_hit_rate ← cache.hit_rate
       // State 2:  Adjust load_admit
7      while true do
8          r ← load_admit
           // Measure target metric at r − step, r, and r + step
9          max_f ← max(f(r−step), f(r), f(r+step))
10         if f(r−step) = max_f then
11             load_admit ← r − step
12         else if f(r+step) = max_f then
13             load_admit ← r + step
14         else
15             load_admit ← r
16         end
17         if load_admit = 0.0 then
18             break                    // Quit tuning when w < w_0
19         end
20         if cache.hit_rate < (1 − α) start_hit_rate then
21             break                  // Workload locality changed
22         end
23     end
24 end
```

rate later changes significantly, indicating that the workload has shifted, the scheduler falls back to State 1 to let the cache re-warm.

NHC is compatible with all common cache write policies, though each interacts differently with read redirection. With write-through, both copies of a block stay clean, so read hits can always be redirected to either device;

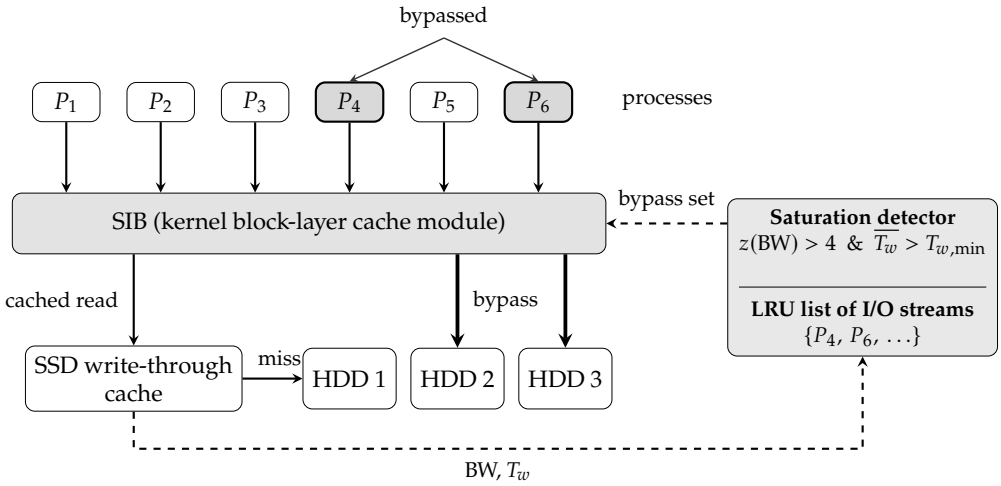


Figure 3.3: SIB Architecture. *The SIB module in the kernel block layer routes each process's reads either through the SSD write-through cache (regular path) or directly to an HDD (bypass). A saturation detector monitors SSD bandwidth (*BW*) and average await time (T_w); when the bandwidth z-score exceeds* 4 *and T_w is above its observed minimum, the SSD is declared saturated. SIB then selects entire processes from its LRU list of large, sequential I/O streams (here P_4 and P_6) and redirects their reads to idle HDDs. Selection is per-process: once a process is in the bypass set, all of its reads avoid the SSD until the process terminates.*

write-through, however, limits write throughput to that of the slower device, since each write must complete on both. With write-back and write-around, only the performance device holds the latest copy of a dirty block, which breaks the two-clean-copy condition that NHC relies on to redirect reads.

3.1.3 Previous Cache-Bypass Approaches

In this section, we describe the two previous state-of-the-art cache-bypass systems we use as baselines: SIB and LBICA.

SIB. SIB (Selective I/O Bypass) [59] was designed for HDFS clusters containing many SSDs and HDDs, where the aggregate HDD throughput is non-trivial. SIB configures SSDs as a write-through cache and proposes directing additional read traffic to the idle HDDs when the SSDs are saturated. The major challenge is detecting SSD saturation: because of internal paral-

lelism, the standard Linux disk utilization metric does not accurately reflect SSD load. SIB therefore estimates saturation by jointly tracking the SSD's bandwidth and average I/O await time across a sliding window, filtering out short-lived fluctuations caused by background activity (e.g., garbage collection). SIB operates in two phases: a *training phase*, in which it measures the SSD's peak bandwidth and corresponding await time for the current workload, and an *inference phase*, in which it compares live metrics against the learned peak and, if the SSD is saturated, redirects reads to the HDDs. SIB makes this redirection decision at *per-process* granularity: once saturation is detected, entire processes (chosen by SIB) are excluded from the SSD cache and send all of their I/O directly to the HDDs.

Figure 3.3 shows the high-level design of SIB. The SIB module sits in the kernel block layer between processes (P_1–P_6) and the storage stack and routes each process's reads either through the SSD write-through cache (regular path) or directly to an HDD (bypass). The control panel on the right captures SIB's offline-trained logic: the saturation detector ingests SSD bandwidth and average await time and flags saturation when the bandwidth z-score exceeds 4 and the await time exceeds its observed minimum. When saturated, SIB consults its LRU list of large, sequential I/O streams and selects whole processes for bypass (here P_4 and P_6, shaded gray); their reads (the bold arrows) flow directly from SIB to the HDDs, while every other process's reads continue through the SSD cache. A process stays in the bypass set until it terminates.

As we show later in §3.3, SIB has three key limitations. First, its training target is inherently unstable: the peak performance of modern devices depends on many workload parameters, such as the read/write ratio, concurrency, and access pattern. Any shift in the workload therefore forces SIB to either retrain or operate with a stale threshold, causing throughput to degrade sharply on dynamic workloads. Second, its per-process bypass granularity is too coarse for general-purpose caching: entire processes are excluded from the SSD cache rather than individual requests being routed, which wastes cache capacity when only a subset of a process's I/O is cold.

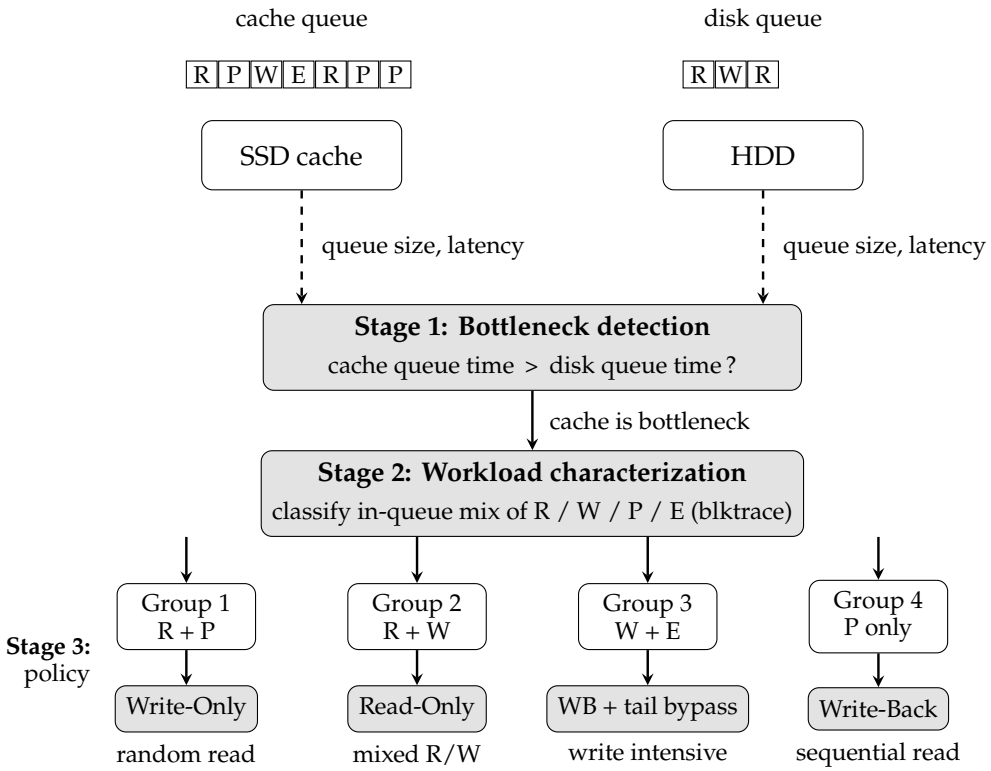


Figure 3.4: LBICA Operation. *LBICA periodically samples the cache and disk queues, then runs a three-stage pipeline.* ***Stage 1*** *detects whether the SSD cache is the bottleneck by comparing the queue time of the cache against that of the disk; queue time is defined as the number of pending requests in the queue multiplied by the device's average request latency, and the cache is declared the bottleneck when its queue time is larger than the disk's.* ***Stage 2*** *classifies the in-queue requests by their mix of read (R), write (W), promote (P), and evict (E) operations into one of four groups.* ***Stage 3*** *switches the cache write policy accordingly: Write-Only for random-read workloads, Read-Only for mixed read/write, Write-Back with tail bypass for write-intensive workloads, and plain Write-Back for sequential reads where the cache has no impact. LBICA does not redirect any read hits.*

Third, SIB does not admit read misses into the cache, further limiting its applicability outside the HDFS setting for which it was originally designed.

LBICA. LBICA (Load-Balancing I/O Cache Architecture) [20] takes a different approach: rather than redirecting individual requests, LBICA adaptively switches the cache's write policy in response to burst load. LBICA

monitors the SSD cache queue and, when it detects that the cache has become the performance bottleneck (burst detection), it adjusts the write policy. For example, it may switch from write-back to read-only or write-only to reduce the load imposed by promotion and eviction traffic. The intuition is that under burst conditions the promote/evict traffic inflates the cache queue far beyond what the hit-rate benefit justifies, so suppressing these background writes improves performance.

Figure 3.4 describes LBICA's three-stage pipeline. Stage 1 samples the queue size and average latency of both the SSD cache and the HDD, computes the maximum queue time of each ($|Q| \cdot L$), and declares the cache as bottleneck when its queue time exceeds the disk's. When the cache is the bottleneck, Stage 2 uses `blktrace` to classify the in-queue requests by their mix of read (R), write (W), promote (P), and evict (E) operations into one of four workload groups: random read (R+P), mixed read/write (R+W), write-intensive (W+E), and sequential read (P only). Stage 3 then switches the cache's write policy accordingly: Write-Only for random reads (admit no reads), Read-Only for mixed read/write (bypass writes to disk), Write-Back with tail bypass for write-intensive workloads, and plain Write-Back for sequential reads where the cache plays no role.

As we show later in §3.3, LBICA has two key limitations. First, it does not redirect read hits to the capacity device: when the cache saturates, LBICA only stops admitting new data, so requests that already hit in the cache continue to queue on an overloaded SSD and the capacity device's bandwidth is left unused. Second, its fixed bottleneck threshold does not adapt to the workload-dependent behavior of modern devices; the right threshold differs across read/write ratios and concurrency levels, so a single static value either misses real bursts or triggers spuriously.

Summary. Both SIB and LBICA have their own limitations: SIB has a stationary algorithm that is hard to adapt to dynamic workloads and does not admit read misses into the cache, while LBICA only controls write admission and does not redirect read hits to the capacity device. NHC, as described above, addresses both issues through a lightweight, continuous feedback

mechanism that redirects read hits at per-request granularity.

3.2 Implementation

We implement NHC as Orthus-CAS, a kernel module within the Open CAS block-layer caching framework [6]. We also re-implement the two previous state-of-the-art cache-bypass systems (SIB and LBICA) as baselines within the same framework, and extend Orthus-CAS to support two additional optimization targets: average latency and P99 tail latency. The rest of this section first introduces Open CAS and walks through how Orthus-CAS plugs into it (§3.2.1), then describes our baseline implementations and the target metrics (§3.2.2).

3.2.1 Open CAS and Orthus-CAS

Open CAS [6] is an open-source, block-layer caching framework developed by Intel. It sits between an application and one or more block devices and uses a configured cache device (typically a faster SSD) to accelerate accesses to a larger capacity device (typically a slower SSD or HDD). Figure 3.5 shows its architecture. A user application issues I/O through the VFS and a chosen logical file system (e.g., ext3 or XFS); the Open CAS Top adapter receives the resulting block requests and forwards them to the Open CAS Framework (OCF), which looks up each request's address in its cache metadata and decides whether to serve it from the cache or forward it to the capacity device. OCF then dispatches through the Bottom adapter to the standard Linux block layer and device driver layer, which split traffic between the backend (capacity) device and the caching device. Cached blocks are managed by a configurable replacement policy (LRU by default), and Open CAS supports the three common write-allocation policies: *write-through* (writes go to both devices and the cache copy is always clean), *write-around* (writes bypass the cache entirely and go to the capacity device, so cached copies are invalidated), and *write-back* (writes are absorbed by the cache and lazily flushed, so cached

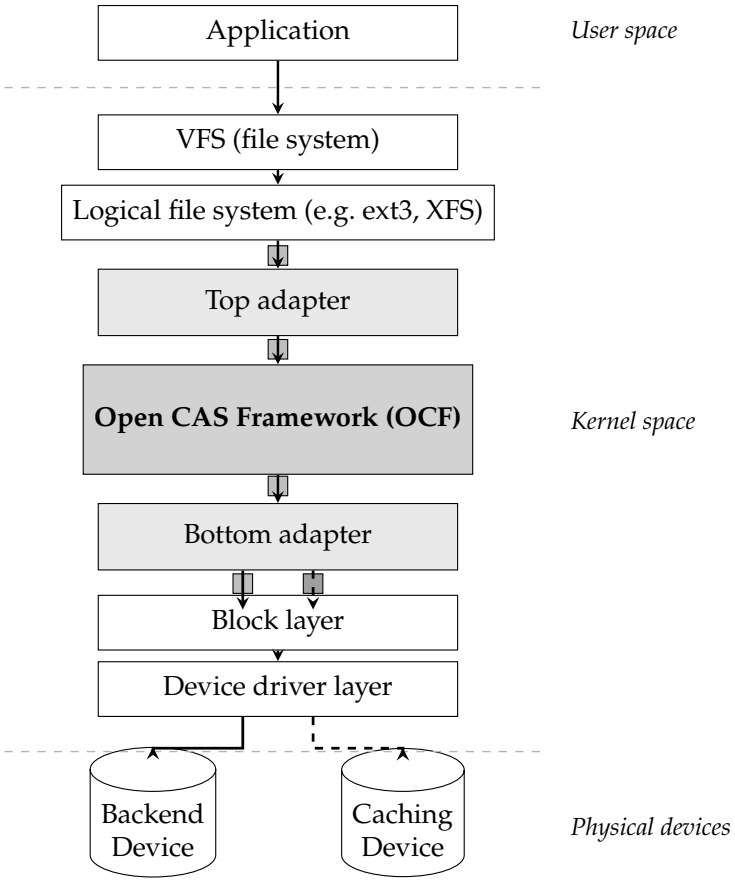


Figure 3.5: Open CAS Architecture. *Open CAS is a kernel block-layer caching framework. User applications issue I/O through the VFS and a chosen logical file system. The Top adapter forwards block requests into the Open CAS Framework (OCF), which decides whether each request hits the cache, then dispatches to the Bottom adapter. The Bottom adapter splits I/O into two paths through the standard Linux block and device-driver layers: backend (blue) reaches the slower capacity device, and caching (orange) reaches the faster cache device. NHC's routing hook and scheduler live inside OCF.*

copies become dirty). Open CAS also exposes per-device utilization counters (bandwidth, queue depth, I/O wait time) that bandwidth-aware schedulers can read to estimate device load.

Orthus-CAS. We implement NHC as Orthus-CAS, a kernel module that lives inside OCF. We extend Open CAS in two ways. First, we add an NHC cache

scheduler that exposes the `data_admit` and `load_admit` knobs described in §3.1.2.3 and runs the two-stage algorithm from §3.1.2.4. Second, we insert a routing hook on the cache read path so that a hit can be dynamically redirected to the capacity device according to `load_admit`. No changes are required to Open CAS's replacement policy or metadata management. Orthus-CAS adds 460 lines of code on top of Open CAS excluding cache-mode registration code.

Write-allocation policies. Open CAS supports write-around, write-back, and write-through; NHC works with all three.

Detecting hit-rate stability. The scheduler treats the hit rate as stable (Algorithm 1, §3.1.2.4) when it changes by less than 0.1% over the last 100 ms. This simple heuristic is sufficient because NHC does not require precise stability detection: under heavy load, a higher measured hit rate just lets NHC redirect more hits, and under light load NHC quickly reverts to classic caching.

Tuning parameters. The scheduler measures the target metric on a 5 ms interval, the smallest interval at which the Linux block-layer counters are accurate, and adjusts `load_admit` by 2% per step. Smaller intervals do not improve convergence noticeably; larger step sizes converge faster but settle on slightly worse operating points.

3.2.2 Baseline Implementations

To enable a fair comparison, we implement both previous state-of-the-art systems within the same Open CAS framework. All baselines share Orthus-CAS's routing hook, write-allocation defaults, replacement policy (LRU), and per-device utilization counters; they differ only in the scheduler that decides where each request goes. Because SIB was originally designed for distributed storage at per-process granularity, we adapt it in two ways: SIB+ moves bypass decisions to per-request granularity, and SIB++ additionally admits read misses to the cache. We describe each baseline below.

SIB+. We faithfully implement SIB in Open CAS following the original paper. SIB+ is an optimized version that bypasses traffic at per-request granularity instead of per-process.

SIB++. SIB++ is an optimized version of SIB+ that additionally admits read misses into the cache.

LBICA. We faithfully implement LBICA in Open CAS following the original paper. LBICA has two key characteristics. First, it detects when the performance device is under "burst load" and stops allocating new data to it. Second, it does not redirect any read hits.

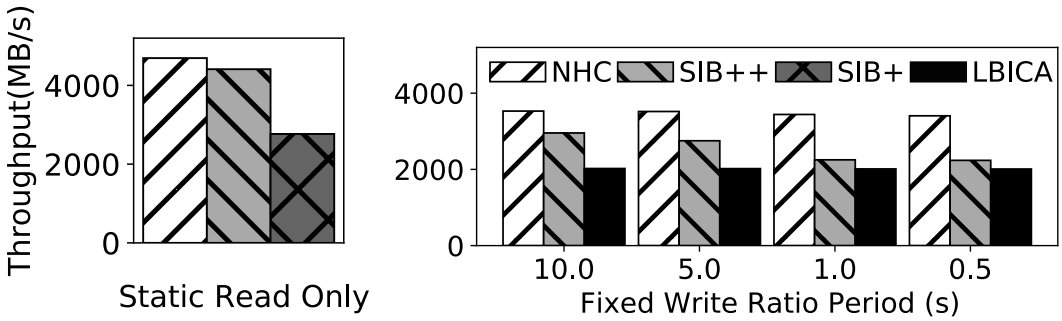
3.3 Evaluation

We evaluate Orthus-CAS against the two previous state-of-the-art systems (SIB+/SIB++ and LBICA) described in §3.1. Our goal is to answer two questions: (i) How does NHC compare against the state-of-the-art baselines under static and dynamic workloads? (§3.3.1) (ii) Can NHC optimize for different performance targets, such as throughput, average latency, and tail latency? (§3.3.2)

Experimental Setup All experiments run on a single server with an Intel Xeon Gold 5218 CPU (16 cores) at 2.3 GHz and 64 GB of DRAM, running Ubuntu 18.04. The storage hierarchy consists of two devices: an Intel Optane SSD as the performance tier and a Samsung NVMe Flash SSD as the capacity tier.

3.3.1 Comparison to Previous Systems

On the Optane+Flash hierarchy, we begin with a read-only workload (Figure 3.6(a)). SIB+ performs poorly because it does not admit read misses into the Optane cache, so a meaningful fraction of requests miss and are served by the slower Flash device even when the cache has free capacity. SIB++ admits read misses and performs much better, approaching NHC under these stable conditions. SIB++ suffers, however, when the workload



(a) Static Workload **(b) Dynamic Workloads**

Figure 3.6: NHC vs. SIB and LBICA. *(a) Static read-only workload. (b) Dynamic workloads where the write ratio changes periodically (e.g., 10 s), varying randomly between 0% and 5%.*

changes (Figure 3.6(b)); in these workloads the write traffic changes every period, for periods between 10 s and 0.5 s. SIB cannot handle dynamic workloads because of its two-phase design: in the training phase, SIB learns the maximum performance of the caching device for the current workload, and in the inference phase, it judges whether the caching device is saturated and, if so, bypasses part of the load. As discussed in §3.1.1, the maximum performance of modern devices depends on many workload parameters, such as the read/write ratio, load, and access pattern. Thus, whenever the workload changes, SIB must either relearn its target performance or operate with an inaccurate one; in our experiments, as the duration of each phase decreases, SIB's performance drops sharply. Unlike SIB, NHC uses a simple, continuous feedback-based tuning approach, so it converges rapidly and adapts well to dynamic workloads. Finally, LBICA performs poorly because it never redirects read hits to the capacity device; it simply stops allocating new data to the performance device when it is overloaded.

3.3.2 Target Performance Metrics

In this section, we show that NHC can optimize for different performance targets. We make three observations from Table 3.2. First, the throughput and average-latency targets converge to nearly the same operating point:

Target	Optane + Flash			NVM + Optane SSD		
	Thru. (GB/s)	Avg lat. (μs)	P99 lat. (μs)	Thru. (GB/s)	Avg lat. (μs)	P99 lat. (μs)
Classic	6.7	77	115	2.3	227	269
NHC, throughput	**8.0**	64	147	**3.9**	132	289
NHC, avg. latency	**8.0**	**64**	143	**3.9**	**132**	285
NHC, P99 latency	6.9	75	**106**	3.3	155	**245**

Table 3.2: Different Target Metrics. *Orthus-CAS performance when configured with throughput, average latency, or P99 tail latency as the target metric. Bold marks the best Orthus-CAS result for each metric in each hierarchy.*

both raise throughput from 6.7 to 8.0 GB/s and drop average latency from 77 to 64 μs on Optane+Flash, with similar gains on NVM+Optane. Second, optimizing for throughput or average latency *increases* tail latency (from 115 to 147 μs on Optane+Flash, from 269 to 289 μs on NVM+Optane); routing more hits to the capacity device exploits its bandwidth but pushes those requests onto a device with worse P99 latency. Third, when configured with P99 latency as the target, Orthus-CAS *reduces* tail latency relative to Open CAS—from 115 to 106 μs on Optane+Flash, and from 269 to 245 μs on NVM+Optane—by deliberately routing fewer hits away from the performance device. These experiments show that NHC behaves as expected under each target metric.

3.4 Conclusion

In this chapter, we revisited caching in modern storage hierarchies and proposed *Non-Hierarchical Caching (NHC)*, an approach that dynamically offloads traffic from the performance device to the capacity device when beneficial. NHC is guided by a lightweight feedback scheduler that adapts online. Through practical experiments, we show that NHC outperforms state-of-the-art cache-bypass systems under both static and dynamic workloads, and can be configured to optimize for different performance targets, such as throughput, average latency, and tail latency.

Chapter 4

Getting the MOST with Mirror-Optimized Storage Tiering

Chapter 3 showed how caching can be extended to exploit the capacity device's bandwidth once the performance device saturates. That approach, however, preserves caching's core limitation: it duplicates every hot block across tiers, wasting the performance device's capacity. In this chapter, we turn to tiering, the standard alternative, and ask how tiering can be made to balance load across non-hierarchical storage systems without the costly data migrations that migration-based tiering requires. We introduce *Mirror-Optimized Storage Tiering* (MOST), a new tier-based approach that mirrors a small amount of hot data across tiers to enable load balancing through routing rather than migration. This chapter is based on the paper *Getting the MOST out of your Storage Hierarchy with Mirror-Optimized Storage Tiering*, published in FAST 2026.

In the first part of this chapter, we show that migration-based tiering—the dominant strategy for managing heterogeneous memory and storage—fails to adapt to dynamic workloads. State-of-the-art systems such as Colloid migrate data across tiers to equalize access latency, but migration is both slow and costly: convergence can take hundreds of seconds on storage hierarchies, and the write traffic incurred significantly shortens device lifetime. Mirroring-based approaches (e.g., caching and RAID-1) avoid these migration costs

but either waste capacity or fail on write-intensive workloads. A different approach is required.

Thus, in the second part of this chapter, we propose MOST. MOST combines tiering's space efficiency with mirroring's load-balancing flexibility by mirroring only a small amount of hot data. Under low load, MOST behaves like classic tiering; under high load, it dynamically routes requests among the mirrored copies to offload excess traffic to the capacity device. A simple feedback-driven optimizer adjusts the routing ratio online, requiring no prior workload or device knowledge. We implement MOST as Cerberus, a storage management layer within the widely-used flash cache library CacheLib [2, 23, 79]. Across static and dynamic workloads on two storage hierarchies, Cerberus achieves up to 2.34× higher throughput, up to 75% lower P99 latency, and up to 84% fewer device writes compared to the state of the art.

In the following, we first present background and motivation related to modern storage hierarchies and prior approaches (§4.1). We then describe the design of MOST and its implementation in Cerberus (§4.2). Next, we present our evaluation (§4.3). We discuss extensions (§4.4) and conclude (§4.5).

4.1 Motivation and Background

In this section, we discuss some of the trends in modern memory and storage hierarchies containing multiple heterogeneous devices. We then describe existing management techniques—such as striping, tiering, mirroring, and caching—and analyze their limitations. We conclude by discussing some of the extra challenges of managing storage hierarchies, as opposed to memory hierarchies.

Storage Device	Latency (μs)		Read (GB/s)		Write (GB/s)	
	4K	16K	4K	16K	4K	16K
Optane SSD	11	18	2.2	2.4	2.2	2.2
PCIe 4.0 NVMe Flash SSD	66	86	1.5	3.3	1.9	2.3
PCIe 3.0 NVMe Flash SSD	82	90	1.0	1.6	1.5	1.6
PCIe 4.0 NVMe Flash SSD over RDMA	88	114	1.2	2.7	1.7	2.3
SATA Flash SSD	104	146	0.38	0.5	0.38	0.5

Table 4.1: Device Performance. *Latency measured for a single-thread read workload; bandwidth for a 32-thread workload. Remote device is connected via a 25 Gbps link.*

4.1.1 Multi-Device Hierarchies

Multiple devices [32, 64, 96, 100, 116, 123, 137] are commonly used to serve memory and storage load. Since new devices may be added incrementally over time to increase capacity or performance, these new devices are likely to exhibit different performance characteristics compared to older devices in the system. One natural approach has been to organize memory or storage devices into a *hierarchy* of tiers, with newer, faster devices placed as a cache above slower, possibly larger devices; such hierarchies have been essential to systems since their inception [27].

However, as recent work has shown [50, 123], new device types pose challenges to ordered hierarchies. To simplify our discussion, we focus on a two-layered hierarchy containing two types of devices: a *performance* device and a *capacity* device, where the performance device is more expensive, smaller in capacity, and faster, whereas the capacity device is cheaper, larger, and slower. In a traditional hierarchy (e.g., DRAM on HDD), the performance gap between the two devices is substantial and the performance of the capacity device can be ignored. However, with emerging technologies such as non-volatile memory [7], low-latency SSDs [8], NVMe Flash SSDs [5, 15], SATA Flash SSDs [14], and remote storage with NVMeoF [13], disaggregated SSDs [60, 61], and EBS [1, 136], the performance and capacity devices have overlapping characteristics. As such, arranging these devices into a strict hierarchy [50, 123] limits their potential.

Table 4.1 shows that latency and bandwidth can be similar for the

performance and capacity devices given current technology. Memory and storage systems can be composed of any pair of these devices. Importantly, the performance ratio across the two tiers may be close: the bandwidth ratio for 16 KB reads is only 1.5:1 between Optane and PCIe 3.0 NVMe devices and only 1.25:1 between local and remote PCIe 4.0 NVMe devices. More subtly, the performance ratios are not static and strongly depend on workload; for example, given 4 KB reads, the ratio between Optane and PCIe 3.0 NVMe devices increases to approximately 2.2:1. These performance ratios depend on access sizes, percentage of writes, and concurrency [122].

4.1.2 Existing Approaches

At a high level, classic approaches for managing multi-device memory and storage can be grouped into two categories: approaches that maintain a single copy of each block (i.e., striping, tiering, and exclusive caching), and approaches that may maintain multiple copies of each block (i.e., mirroring, caching). We describe these approaches and highlight their inefficiencies, as summarized in Table 4.2.

Single Copy *Striping* [30, 34, 42, 91, 92, 101] allocates a single copy of each block in a predetermined static pattern across devices. Striping does not handle heterogeneous devices well: if data is striped evenly across devices, throughput is bottlenecked by the slowest device; if data is striped in a weighted pattern with more data on faster devices, the appropriate weight depends on the workload; furthermore, the ideal striping ratio for performance is unlikely to match the ratio of the devices' capacity (i.e., faster devices are usually smaller). Thus, striping is not a good match for heterogeneous storage.

Tiering [18, 37, 63, 77, 94, 105, 108, 121, 128, 137, 139] also allocates a single copy of each block; however, instead of allocating blocks in a static pattern, tiering dynamically places data blocks based on their hotness [18, 63, 76, 94, 108, 128, 137, 139] or some other optimization goal [32, 46, 131]. Table 4.2 summarizes the characteristics of three tiering approaches.

		Single Copy				Multiple Copy		
		Striping	HeMem	BATMAN	Colloid	Mirroring	Orthus	MOST
Bandwidth Utilization	Random Read-only	Low ●	Low ●	Medium ●●	Medium ●●	High ●●●	High ●●●	High ●●●
	Random Write-only	Low ●	Low ●	Medium ●●	Low ●	Low ●	Low ●	High ●●●
	Random RW-mixed	Low ●	Low ●	Medium ●●	Medium ●●	Medium ●●	Medium ●●	High ●●●
	Sequential Write	Low ●	Low ●	Low ●	Low ●	Low ●	Low ●	High ●●●
Capacity Utilization		High ●●●	High ●●●	High ●●●	High ●●●	Low ●	Low ●	High ●●●
Dynamic Workload		Low ●	Low ●	Low ●	Low ●	Medium ●●	Medium ●●	High ●●●

Table 4.2: Qualitative Comparison of Different Techniques in a Modern Storage Hierarchy. *Bandwidth utilization is categorized as follows: Low means no load-balancing mechanism; Medium represents limited load-balancing with restrictions under specific workloads; High indicates effective load-balancing across a wide range of workloads.*

First, classic hotness-based tiering is exemplified by HeMem [96], which manages a memory system containing DRAM and NVM. HeMem promotes hot data into the performance tier and serves hot data exclusively from the performance tier, leading to inefficient utilization of the capacity tier's bandwidth; thus, HeMem delivers low bandwidth for intensive workloads due to not effectively utilizing NVM (as shown later in §4.3). Second, BATMAN [32] balances tier bandwidth by migrating data, but its fixed bandwidth ratio prevents it from adapting to hierarchies whose performance ratios vary with workload or evolve over time. Third, Colloid [116] also dynamically migrates data between memory tiers to utilize the bandwidth of the capacity device; however, Colloid does not handle dynamic and time-varying workloads because costly data migrations are required to adjust the load distribution when the workload changes (§4.3.2).

Other tiering systems, such as AutoTiering [131] and EDT-DTM [46], also balance data placement to utilize the bandwidth of all tiers, but since they rely solely on data migration to adjust load distribution, their adaptability and responsiveness to workload changes are limited. *Exclusive caching* [44] maintains only a single copy of data within the hierarchy: when hot data is promoted or demoted between tiers, the original copy is discarded. At a high level, exclusive caching is similar to hotness-based tiering but moves data at smaller time intervals; consequently, it behaves similarly. In summary, as shown in Table 4.2, single-copy approaches may partially utilize the bandwidth of the capacity device by migrating data across tiers. However, this migration-based strategy struggles to adapt to dynamic workloads, as migration is both costly and slow. In addition to limited responsiveness, migration-based approaches also suffer from excessive device writes and performance degradation due to migration-induced traffic (see §4.1.3 for further discussion).

Multiple Copies To utilize both devices, other approaches maintain two copies of at least some data blocks across both the performance and capacity devices. *Mirroring* [25, 30] simply replicates all data on both devices. Mirroring delivers high read bandwidth since reads can be dynamically

load balanced across both devices to account for performance differences; however, mirroring delivers low write bandwidth since both copies must be updated and bandwidth is limited by the slower device. Mirroring also has low capacity utilization since each block is stored on both devices.

Inclusive caching allocates all items on the capacity device, but replicates only frequently-accessed items on the performance device [6, 23, 40, 79, 88, 89, 93, 123, 129]; inclusive caching inherently wastes the capacity of the performance device and fails to exploit the performance of the capacity device [20, 59]. To utilize the performance of the capacity device, Orthus [123] introduces Non-Hierarchical Caching (NHC). Orthus dynamically redirects read traffic from the performance device to the capacity device when the performance device is overloaded. However, Orthus is built on a traditional caching model, which introduces two fundamental limitations. First, it is space-inefficient—NHC uses the entire capacity of the performance device to store duplicate copies of data from the capacity tier, wasting the performance tier's storage capacity. Second, Orthus struggles with write-intensive workloads, requiring clean copies for routing read requests: writes that hit in the cache can be handled with write-through or write-back/write-around. With write-through, both copies remain clean, but additional writes are incurred, and performance is constrained by the write bandwidth of the capacity device. With write-back/write-around to only one of the block's copies, Orthus can only route subsequent reads to the dirty block. Thus, Orthus performs poorly with writes (§4.3.1).

Nomad [126] proposes a variant to hotness-based tiering that maintains temporary copies of data during migration; specifically, the original copy on the capacity device can still be accessed while the data is being migrated to the performance device. While Nomad improves the performance penalty of migration, it does not maximize the bandwidth of the underlying devices in the common case.

Summary. Existing approaches for handling heterogeneous memory and storage fail along important metrics (Table 4.2). Approaches that maintain a single copy of data, such as striping, tiering (e.g., HeMem, BATMAN, Colloid,

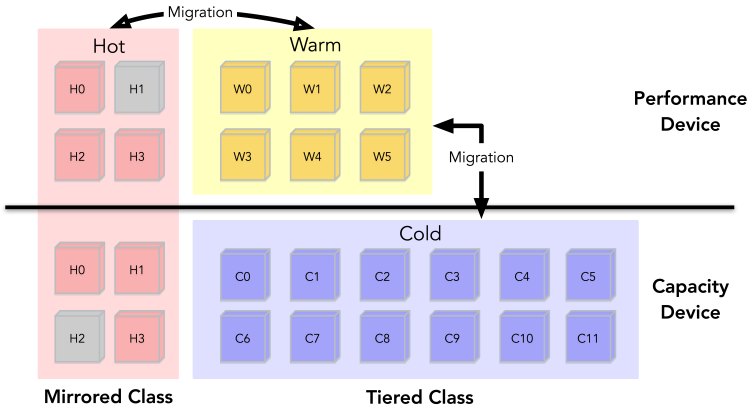


Figure 4.1: MOST Data Layout. *Data are logically grouped but not physically placed together on the device.*

AutoTiering, and EDT-DTM), and exclusive caching, fall short in handling dynamic workloads. Current approaches, such as mirroring, inclusive caching, and non-hierarchical caching, that maintain multiple copies of data items suffer from low capacity utilization and struggle with write-intensive workloads. We will show that MOST delivers high bandwidth and reduces device writes across a wide range of static and dynamic workloads with only a small amount of mirrored data.

4.1.3 Storage vs. Memory Hierarchies

The approaches that handle data placement in memory hierarchies [32, 64, 96, 116] versus storage hierarchies [120, 131, 137] have many similarities. In both the memory and the storage domain, strategies for data placement optimize for many of the same metrics, such as high performance (bandwidth and latency) and high capacity. In both domains, solutions share many of the same high-level approaches (e.g., striping, tiering, caching), as well as similar low-level techniques (e.g., attempting to balance load across heterogeneous devices by reacting to observed performance whether for memory [32, 116] or for storage [123]).

However, storage hierarchies have characteristics that require special attention and introduce additional opportunities.

Larger Capacity and Datasets Storage devices manage significantly larger datasets than memory. Thus, if migration is used as the primary technique for balancing load across tiers, more data will need to be migrated, leading to larger convergence times when adjusting to workload changes.

Limited Write Bandwidth Storage devices have lower write bandwidth than memory; combined with the previous point, this further increases convergence time if migration is used to adjust to dynamic workloads.

Read/Write Interference Some storage devices [122, 124] have complex performance characteristics such that write operations significantly impact the performance of read operations. Migrating data in the background introduces write traffic that can significantly degrade foreground performance.

Device Endurance Frequent migrations accelerate device wear, reducing the lifespan of storage devices such as SSDs [78, 79].

Software-based Indirection Since storage devices have slower access time (10–500 μs) than memory (50–100 ns), software is used to determine the location of blocks on different devices instead of hardware. As a result, techniques such as dynamic and selective mirroring are more straightforward to implement for the storage stack.

Thus, while dynamic tiering with data migration has been shown to work well for heterogeneous memory hierarchies [96, 116], heterogeneous storage hierarchies need additional techniques that do not rely as heavily on migration.

4.2 Mirror-Optimized Storage Tiering

We present *Mirror-Optimized Storage Tiering* (MOST), a combination of mirroring and tiering that enables flexible load balancing across tiers by mirroring a small amount of data. MOST is based on three design goals:

Maximized Bandwidth Utilization MOST should dynamically balance load across all storage tiers based on workload.

Quick Response to Dynamic Workloads MOST should dynamically and quickly adapt to varying workloads, including fluctuations in load and working set size.

Independence from Device and Workload Characteristics MOST should operate effectively across any storage hierarchy without requiring prior knowledge of device characteristics or workload patterns.

4.2.1 Basic Design

The key idea of MOST is to combine the load-balancing advantages of mirroring with the capacity benefits of classic tiering. MOST uses a hybrid data layout with two classes: the mirrored class and the tiered class (Figure 4.1). Data in the mirrored class is replicated across two devices to enable fast load balancing; data in the tiered class is stored as a single copy to maximize the space efficiency. In MOST, *hot* data, which is accessed most frequently, is moved to the mirrored class and the remaining data is stored in the tiered class. Within the tiered class, *warm* data is migrated to the performance device; the remaining *cold* data is migrated to the capacity device.

As desired, the performance device stores the most-accessed data: *hot* data in the mirrored class and *warm* data in the tiered class. Under light workloads, this layout maximizes the likelihood that a request can be served from the performance device. Under intense workloads, MOST dynamically increases the size of the mirrored class and routes some requests in the mirrored class to the capacity copy to balance the load across both devices.

4.2.2 Design Details

Figure 4.2 shows the basic architecture of MOST. Blocks are placed in the mirrored or tiered class depending on their frequency of access: the goal is to place the hottest data in the mirrored class, warm data in the tiered class on the performance device, and cold data in the tiered class on the capacity device. When handling a request, MOST first determines if the request is to the tiered or mirrored class. Requests to tiered data are straightforward since

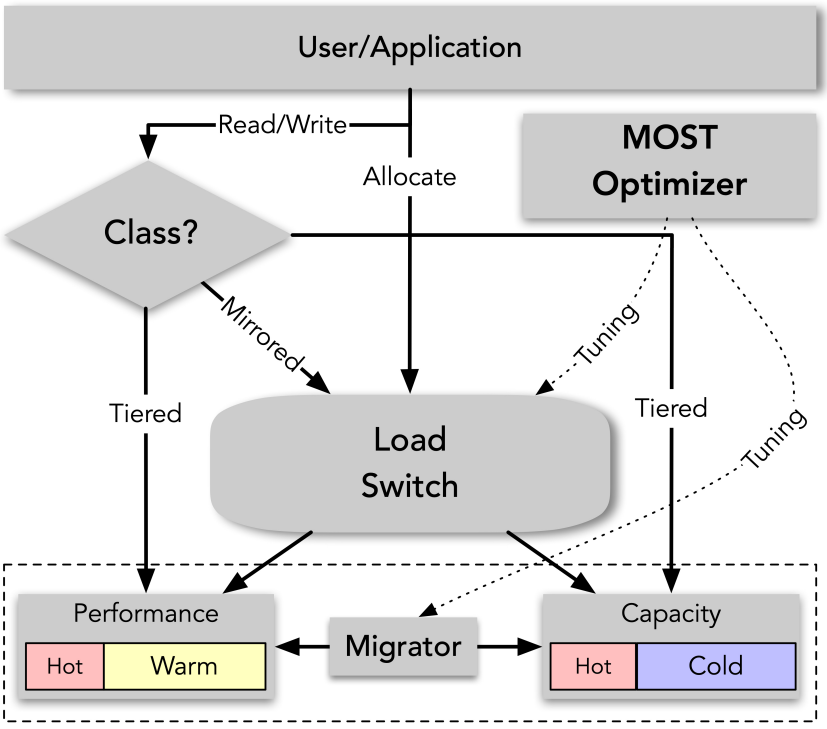


Figure 4.2: MOST Architecture.

the data resides strictly in either the performance or capacity device; those reads and writes are simply forwarded to the appropriate location. Requests to mirrored data involve more complexity: a load switch (or balancer) sends some percentage of these requests to each device, where the percentage depends upon the current workload performance and whether the request is a read or write. Other components of the architecture calculate this percentage (i.e., Optimizer) and migrate data between the classes and devices (i.e., Migrator).

We address the key questions of MOST's design. For reads to the hottest data, which data copy in the mirrored class should be read to balance traffic across devices? On which device should newly written data be allocated? What data should be migrated between mirrored and tiered classes, and between devices? For writes to mirrored data, which copy should be updated? At what granularity should clean copies be tracked? When should copies be

Algorithm 2: MOST Optimizer

```
1  while true do
2      sleep(tuningInterval); Measure end-to-end latency
3      if L_P > (1 + θ) · L_C then
4          if offloadRatio = offloadRatioMax then
5              if Mirrored class is not maximized then
6                  Enlarge the mirrored class
7              else
8                  Improve hotness of the mirrored class
9              end
10             Only migrate to capacity device
11         offloadRatio += ratioStep
12     else if L_P < (1 − θ) · L_C then
13         if offloadRatio = 0 then
14             Only migrate to performance device
15         else
16             offloadRatio −= ratioStep
17         end
18     else
19         Stop all migration
20     end
21 end
```

cleaned? How can the tail latency of mirrored data be protected?

4.2.2.1 Balancing Read Traffic in the Mirrored Class

For simplicity, we begin by focusing on reads that are to the hottest data and thus in the mirrored class. We later show how MOST allocates data, moves data to and from the mirrored class, and handles writes. The full algorithm is shown in Algorithm 2.

At a high level, reads to mirrored data should be routed to the device that delivers the best performance. To achieve this, MOST employs probability-based routing to determine which copy to read [123]. Specifically, MOST routes an incoming read with probability *offloadRatio* to the capacity device, and otherwise, to the performance device. *OffloadRatio* is the key variable con-

trolling load distribution, as well as write traffic and allocation. *OffloadRatio* is tuned by a simple yet efficient feedback-driven dynamic optimization algorithm within the Optimizer; *offloadRatio* is adjusted such that the measured end-to-end latency of the devices is equalized.

Specifically, assuming L_P is the latency of the performance device and L_C is the latency of the capacity device, then

- When $L_P < L_C$, accessing the performance device is faster than accessing the capacity device, and so more traffic should be routed to the performance device; thus, *offloadRatio* is decreased.
- When $L_P > L_C$, accessing the capacity device is faster than accessing the performance device, and so more traffic should be routed to the capacity device; thus, *offloadRatio* is increased.
- When $L_P = L_C$, the two access latencies are approximately equal, and therefore, no further action is required.

MOST efficiently adapts to both high-load and low-load scenarios. Under low load, the latency of accessing the performance device is lower than that of the capacity device, and so traffic is routed to the performance device; in these conditions, MOST operates like classic tiering, directing as many requests as possible to the performance device. In contrast, under high load, all hot and warm data reside on the performance device, causing its access latency to increase—due to queuing and internal resource contention—and eventually exceed that of the capacity device; consequently, MOST offloads requests to the capacity device, which improves throughput. Under high load, MOST offloads traffic from the performance device to the capacity device until the end-to-end latency between the two devices is equalized.

4.2.2.2 Dynamic Write Allocation

When a data block is first written, MOST allocates its location. MOST divides both mirrored and tiered storage into fixed-size segments (e.g., 2 MB) and maintains in-memory data structures to track the metadata for the segment. All data are initially allocated into the tiered class. In classic tiering, the

allocation policy is load-unaware, meaning that it always allocates newly-created blocks on the performance device, even when that device is saturated. MOST introduces a probability-based write allocation policy: similar to load balancing within the mirrored class, the newly allocated data is placed on the capacity device with *offloadRatio* probability. As a result, when the performance device is under high load (higher latency), more data is allocated on the capacity device; when the performance device is under light load (lower latency), all data is allocated on the performance device, as desired and as in classic tiering [96, 116]. After a block has been allocated to the tiered class, it may later be promoted to the mirrored class, as described next.

4.2.2.3 Mirror-Class Migration

To meet the basic goal of placing the hottest data in the mirrored class, MOST migrates data between classes such that it minimizes the amount of movement between devices. To identify hot/warm/cold data, MOST tracks read and write counters for each segment, similar to HeMem [96]; thus, hotness is based on access frequency.

When the amount of mirrored data is insufficient to effectively balance traffic, MOST increases the size of the mirrored class up to a configured maximum. Our experiments (§4.3) show that devoting 20% of the total capacity to the mirrored class is sufficient for our workloads. To migrate data into the mirrored class, MOST selects the hottest segment from the tiered class on the performance device; this hottest data is simply duplicated onto the mirrored class on the capacity device. If the mirrored class reaches its maximum size and the hottest segment in the tiered class has a higher access frequency than the coldest segment in the mirrored class, MOST swaps those segments. Space within the mirrored class is reclaimed when the available system capacity drops below a predefined watermark (2.5% of the total capacity). During reclamation, MOST reclaims the coldest segment in the mirrored class: if a valid copy of this segment exists on the performance device, the capacity copy is discarded; otherwise, the copy on the performance device is discarded.

Migration Regulation. In classic tiering, hot data is always migrated to the performance device and cold data to the capacity device; however, with MOST and its mirrored class, hot segments may be migrated to the capacity device. MOST dynamically manages this bidirectional migration, adhering to a principle of migrating exclusively *away* from the device experiencing higher end-to-end latency. Precisely, when the performance device has higher latency, migration to the performance device is stopped and migration to the capacity device is enabled; when the capacity device exhibits higher latency, the opposite is performed; if the latencies of both devices are approximately equal, all migration is stopped.

4.2.2.4 Balancing Writes and Tracking Clean Subpages

Our previous discussion focused on reads; writes need special consideration. Writes to the tiered class remain straightforward since there is only one copy to update. However, writes to the mirrored class have a new property: if the write is performed to both copies, then the system is not performing any write load balancing; however, if the write updates only one copy, then future reads must be directed only to that clean, valid copy. In order to perform load balancing of writes, MOST updates only one copy and carefully tracks which portions of each segment are valid.

Writes to the mirrored class are balanced as follows. If both copies of the mirrored data are valid, the write request is probabilistically sent to either the capacity or performance device based on *offloadRatio*. However, if only one copy in the mirror is valid, a naive implementation based only on segments must direct later write traffic only to the valid segment (or overwrite the entire invalid segment, making it valid).

Mirrored Data Subpages. To enable better load balancing of writes, MOST manages segments in the mirrored class at a finer granularity: an invalid bit and a location bit are tracked for each subpage corresponding to the device's unit of access (e.g., 4 KB). Each subpage exists in one of three states: clean (both copies are valid), invalid on the performance device, or invalid on the

capacity device. Thus, given an aligned subpage write to the mirrored class, MOST can route the write to either device without having to update the entire segment; that is, a 4 KB-aligned write can be load balanced through simple routing, similar to reads. Subpages in the mirrored class slightly increase metadata overhead (2 bits of metadata), but since the mirrored class is relatively small, the overall overhead remains minimal; for instance, in a 2 TB hierarchy in the extreme case where all performance device data is mirrored (50% mirroring), the metadata overhead is only 128 MB, which is negligible.

Selective Cleaning in the Mirrored Class. Data blocks with only one valid copy are selectively cleaned by a background thread. This cleaning thread selects blocks with a large *rewrite distance*, the average number of reads between two writes for a given block. We have observed that when a block has a small *rewrite distance*, it is likely to be rewritten soon, making cleaning ineffectual.

4.2.2.5 Tail Latency Protection

The description thus far has focused on maximizing total bandwidth from the performance and capacity devices; however, MOST allows users to protect the tail latency of mirrored (hot) data by setting a maximum *offloadRatio* which limits the traffic offloaded to the capacity device in the mirrored class when the capacity device shows significantly worse tail latency than the performance device.

4.2.3 Implementation: Cerberus

We introduce Cerberus, a user-level storage management system that integrates MOST into CacheLib. CacheLib is a generic flash cache library used extensively in data center infrastructures [12, 23, 38, 79, 109, 111]. As shown in Figure 4.3, CacheLib consists of three layers: a DRAM cache layer, a flash cache layer, and a storage management layer. CacheLib provides two default flash cache layers: a Large Object Cache (LOC) employs a log with a DRAM

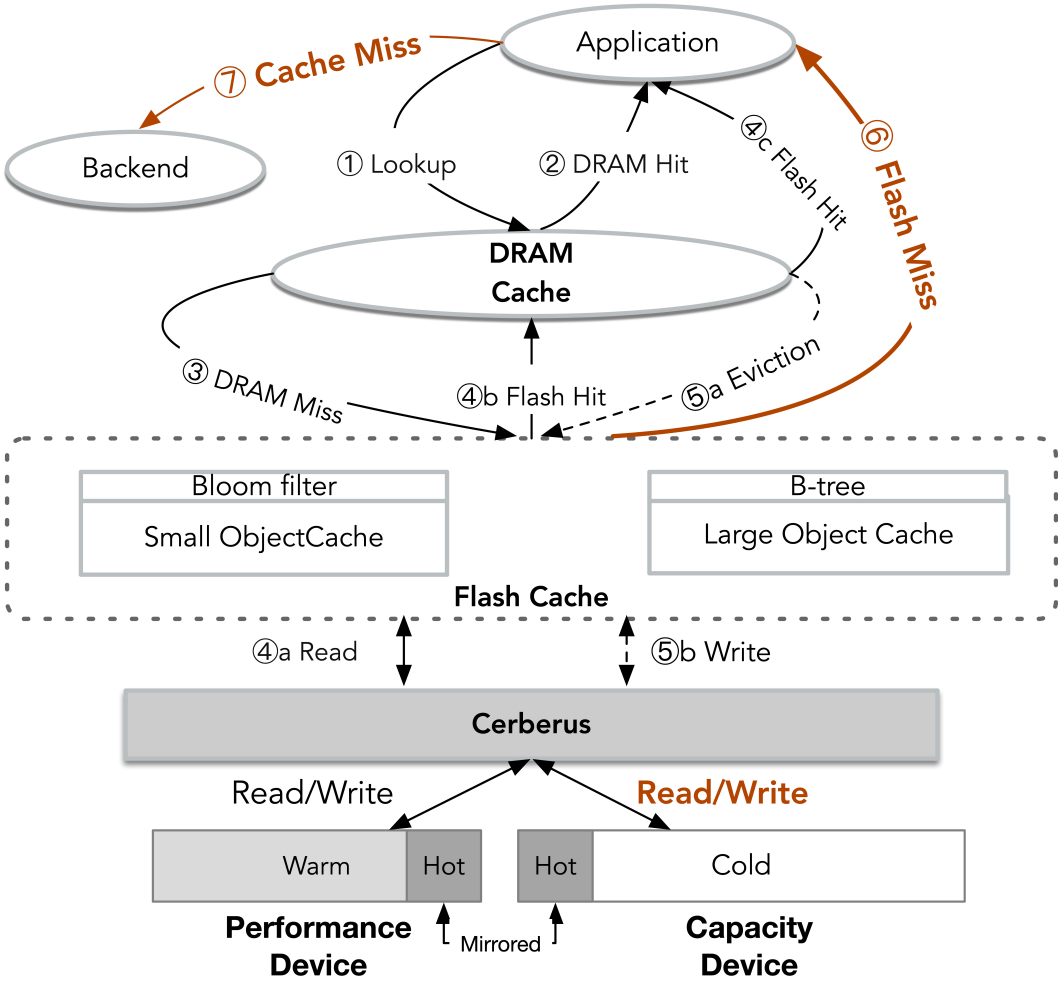


Figure 4.3: CacheLib Architecture. *This figure shows CacheLib's architecture and the lookup workflow. A lookup first checks the DRAM cache (①) and immediately returns the object on a hit (②). On a miss, it checks the flash cache (③), issuing a read to the underlying storage devices (④a); if the object is found, it returns the result (④b). A flash cache hit promotes the item to DRAM and may evict an existing DRAM entry (⑤a, ⑤b). A miss in the flash cache (⑥) leads to a back-end access (⑦).*

index for items 2 KB or larger; a Small Object Cache (SOC) stores key-value pairs in a 4 KB bucket hash table. Users can also implement custom flash caches in this flash cache layer (e.g., Kangaroo [79] and Fairy-WREN [78]).

By default, the storage management layer in CacheLib only provides striping across devices. Cerberus is our storage management layer between

the flash cache engine and the storage hierarchy employing MOST; Cerberus transparently manages the underlying performance and capacity devices, providing a block interface and a large address space. In addition to MOST, we have implemented other tiering and caching approaches within the CacheLib storage management layer.

Striping The default implementation in CacheLib.

Orthus The non-hierarchical caching implementation described and provided by Wu et al. [123] in 0.6K lines of code.

HeMem Our implementation of the classic tiering and migration algorithm [96] in about 0.7K lines of code. The original HeMem uses a quantum of 10 ms, which is appropriate for memory latency but not storage; we use 200 ms, the smallest interval that accurately reacts to our storage device latency.

BATMAN Our implementation of BATMAN heterogeneous tiering [32], requiring about 0.4K lines of code.

Colloid Our implementation of Colloid [116] is built on HeMem (with a 200 ms quantum) in about 0.4K lines of code. Since Colloid only balances reads, we implement Colloid+ to incorporate write latency into decisions. Additionally, since Colloid is sensitive to parameters, we implement Colloid++ with $\theta = 0.2$ and $\alpha = 0.01$, which improves robustness given storage device performance fluctuations.

Metadata Management. MOST divides storage into 2 MB segments, where each segment requires 76 bytes of metadata as shown in Table 4.3. Smaller segments would result in larger metadata overhead (e.g., 4 KB segments require 512 times more metadata) and degrade performance since they obtain lower device bandwidth (Table 4.1). Conversely, larger segments would lead to inefficient utilization of the performance tier, as only a small fraction of subpages might be frequently accessed. We have found that 2 MB segments balance the metadata footprint while minimizing space waste on the performance tier, matching the choice of many other systems [64, 96, 116].

Member Variable	Size (bytes)
id (uint64_t)	8
addr[2] (uint64_t[ ])	16
invalid (bitset<512>*)	8
location (bitset<512>*)	8
clock (uint64_t)	8
readCounter (uint8_t)	1
writeCounter (uint8_t)	1
rewriteReadCounter (uint64_t)	8
rewriteCounter (uint64_t)	8
flags (uint8_t)	1
storageClass (enum class)	1
mutex (SharedMutex)	8
Total	**76**

Table 4.3: In-Memory Metadata per Segment.

Implementation Details. The MOST optimizer runs on a single pinned thread during each 200 ms interval. At each interval, the optimizer estimates the access latency of each device by comparing counters from the Linux block layer [11] to measurements from the previous interval. Similar to prior systems [116], we apply Exponentially Weighted Moving Average (EWMA) to measured latency to smooth out short-term fluctuations and maintain long-term stability. The optimizer leverages this smoothed latency signal to guide migration decisions and adjust routing accordingly. We set $\theta = 0.05$, a commonly used tolerance in tuning-based systems [116], to treat values as approximately equal, striking a balance between stability and responsiveness. MOST demonstrates robust performance across diverse workloads without requiring fine-tuning, indicating low sensitivity to the specific choice of θ. We also adopt $ratioStep = 0.02$, following guidance from similar systems (e.g., Orthus [123]), which works well across different workloads. Cerberus introduces approximately 1.5K LOC to CacheLib, leveraging and extending the core HeMem tiering logic.

4.3 Evaluation

We evaluate Cerberus to answer the following questions.

- How does Cerberus compare to striping, caching (Orthus), and tiering (HeMem, BATMAN, and Colloid) for static workloads with different intensity levels and patterns? (§4.3.1)
- How quickly does Cerberus adapt to dynamic workloads? How does the number of writes performed by MOST compare to migration-based tiering approaches with load-balancing capabilities? (§4.3.2)
- How effective are Cerberus's techniques for mirrored class sizing, subpage management, and selective cleaning? (§4.3.3)
- How does CacheLib with Cerberus perform on static workloads? How does it perform on dynamic workloads? (§4.3.4)

Storage Configurations. We evaluate two storage hierarchies: Optane (performance) / NVMe (capacity), and NVMe (performance) / SATA (capacity). The latency and bandwidth of these three storage devices (750 GB Intel Optane SSD P4800X, 1 TB Samsung 960 NVMe SSD, 1 TB Samsung 870 SATA SSD) are shown in Table 4.1. The server has a 40-core Intel Xeon Gold 5218R CPU @ 2.1 GHz (Ubuntu 20.04) and 64 GB DRAM.

4.3.1 Comparison to Previous Approaches

We begin with a high-level comparison of Cerberus to other approaches for handling heterogeneous devices: striping, Orthus (caching), HeMem (classic tiering), BATMAN (with a static ratio matching the read bandwidth of the devices), and three versions of Colloid (a state-of-the-art tiering approach). To focus on the basic performance of these algorithms, we isolate the storage management layer from CacheLib and exercise that layer with controlled workloads; we begin with a static micro-benchmark in which multiple threads perform synchronous random reads and writes, given a skewed access pattern where a 20% hotset is accessed with 90% probability. Figure 4.4 compares steady-state throughput for read-only, write-only, sequential write, and read-

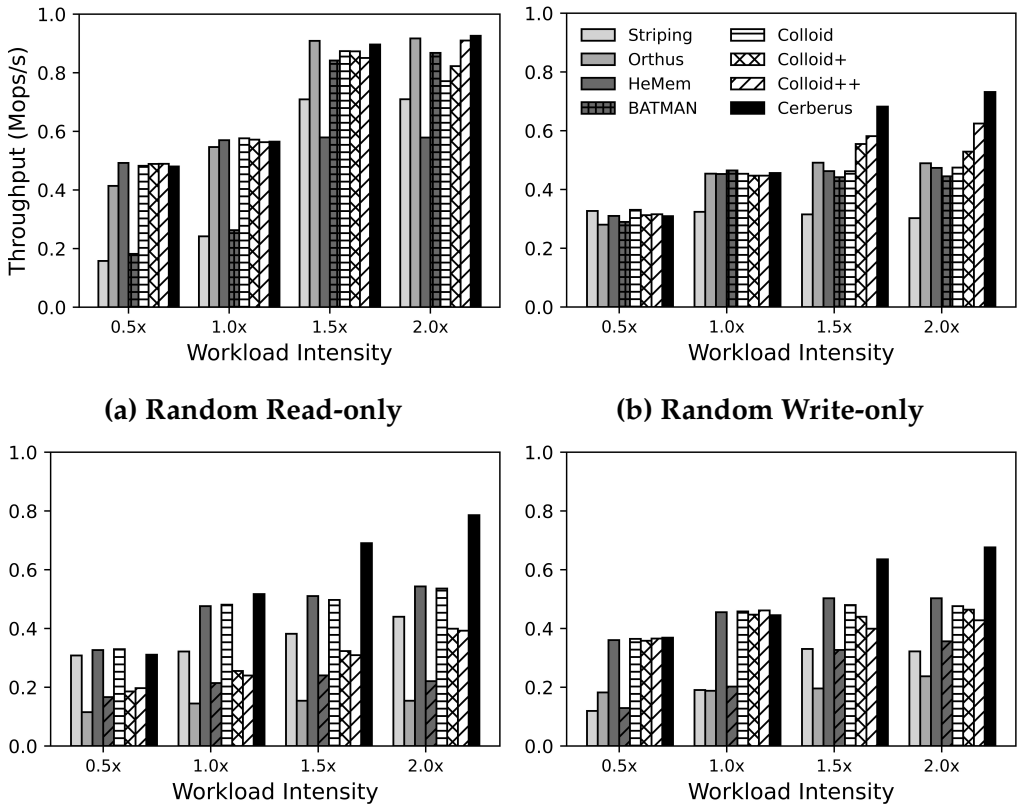


Figure 4.4: Static Workload. *The workload is running on the Optane/NVMe hierarchy with 750 GB working set. In (a), under intensity 2.0×, Colloid, Colloid+, Colloid++, and Cerberus migrate 134 GB, 122 GB, 62 GB, and 50 GB of data in total. In (b), under intensity 2.0×, Colloid, Colloid+, Colloid++, and Cerberus migrate 0 GB, 93 GB, 80 GB, and 65 GB of data in total.*

latest workloads with varying intensities; 1.0× represents the minimum load at which the bandwidth of the performance device is saturated. We show that on static workloads Cerberus performs as well as or better than all other approaches.

Random Read-only. Even on this simple static workload, the other approaches do not match Cerberus. Striping delivers suboptimal performance, since it is bottlenecked by the slower device. Orthus delivers similar throughput to Cerberus, but Orthus achieves this by mirroring 690 GB of data

compared to only 50 GB in Cerberus. HeMem (classic tiering) reaches its performance limit when the load intensity is 1.0× because HeMem does not offload traffic to the capacity device once the performance device's bandwidth is saturated; consequently, further load increases do not improve throughput. For BATMAN, no static allocation ratio works well for all load levels: BATMAN performs relatively well under high load because its allocation ratio is configured to match this bandwidth ratio for the hierarchy; however, at low load, this ratio causes traffic to be sent to the capacity device, reducing throughput. Colloid experiences significant performance degradation at intensity 2.0× due to migrations triggered by latency spikes arising from background activity; *Colloid*++ improves this throughput, showing that Colloid is sensitive to parameter choice. As a result, Colloid and Colloid++ incur 2.68× and 1.24× more migration traffic than Cerberus (as shown in the caption). In summary, Cerberus delivers high throughput at both low and high load and also reduces device writes compared to migration-based approaches that attempt to balance the load.

Random Write-only. We focus on those results that are noticeably different for writes versus reads. Orthus has a static write-back policy, and so does not balance write traffic or scale under high load. BATMAN no longer performs well at high load because a different allocation ratio is required to match the performance of write traffic versus read traffic across the two devices. Colloid does not perform write balancing and therefore exhibits performance similar to HeMem; *Colloid*+ balances write traffic but suffers from migration overhead triggered by latency spikes, resulting in suboptimal throughput; *Colloid*++ demonstrates improved performance, but still incurs migration overhead. For write-intensive workloads, Cerberus performs significantly better than other approaches, since it balances writes in the mirrored class and is robust to latency spikes.

Sequential Write. Sequential writes emulate popular applications with log-structured data (e.g., flash caches, file systems, and databases). As with random writes, Orthus cannot efficiently balance write traffic. Colloid

behaves similarly to HeMem because it doesn't take write latency into account when balancing. Interestingly, *Colloid+* and *Colloid++* perform worse due to additional interfering migrations when balancing write latency; in a sequential workload, demoted blocks are not re-accessed, which makes these migrations ineffective. As desired, Cerberus dynamically allocates new writes proportionally across devices: when the performance device becomes saturated, writes are allocated directly on the capacity device.

Read Latest. This workload uses a 50% write ratio, where 20% of the newly-written blocks have a 90% probability of being read. Colloid performs worse than HeMem at intensities higher than 1.0× due to migration traffic; these migrations are ineffectual since the migrated blocks soon become cold as new blocks are written into the system. Colloid+ and Colloid++ perform more migration, resulting in worse performance. Cerberus efficiently balances the workload by dynamically allocating a portion of new writes to the capacity device when the performance device is saturated; as a result, Cerberus effectively uses the bandwidth of each device as the intensity increases.

Summary. For a variety of static workloads, Cerberus achieves better throughput than striping, Orthus, BATMAN, HeMem, and Colloid. We omit BATMAN in subsequent experiments since it performs worse than Colloid or its variants.

4.3.2 Dynamic Workloads

Dynamic time-varying workloads are challenging for previous approaches. To compare Cerberus to HeMem and Colloid++, we create a micro-benchmark modeling a bursty workload similar to those in practice [23]. After pre-warming the system with intensive load for 1000 seconds, the benchmark transitions to generating a 2-minute burst every 15 minutes. Figures 4.5 and 4.6 show the delivered throughput and amount of data promoted and demoted as a function of load (i.e., thread count) for read-only, write-only, and read-write workloads.

Read-only workload. Before 1000 seconds, the systems operate under

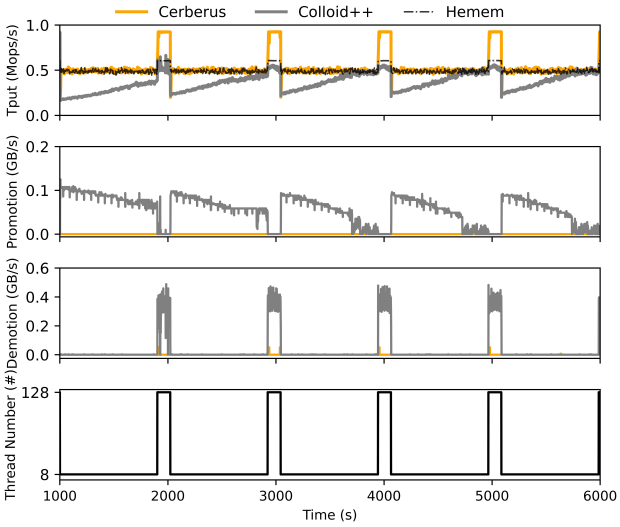
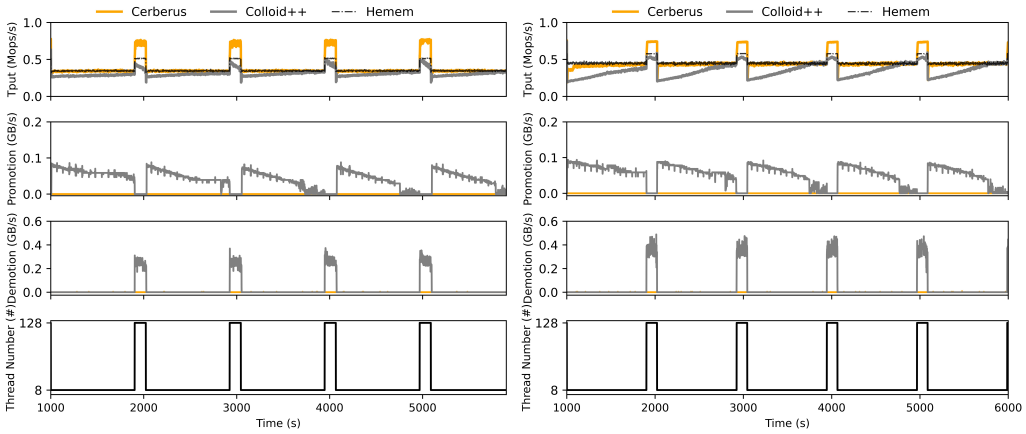


Figure 4.5: Dynamic Bursty Workload: Read-only. *Running on Optane/NVMe hierarchy with 1.2 TB working set and same skewness as §4.3.1. Under the read-only workload, Colloid++ migrates 282 GB to the performance device and 262 GB to the capacity device. In comparison, Cerberus migrates 87 GB of data to the capacity device.*

high load; in this warm-up period, Colloid demotes and Cerberus mirrors approximately 7% of the hot data to the capacity device. At 1000 seconds, when workload intensity drops, Colloid promotes the hotset back to the performance device, causing intensive promotion traffic, which negatively impacts throughput. In contrast, within 10 seconds Cerberus efficiently rebalances traffic by routing requests to mirrored data back to the performance device. When the 2-minute load burst occurs after about 1000 seconds, the performance device is saturated, causing its latency to surpass that of the capacity device. Colloid again demotes the hotset back to the capacity device. Consequently, Colloid performs worse than HeMem, which simply keeps hot data on the performance device and does not conduct any load balancing. Again, during the load burst, Cerberus efficiently offloads excess traffic by redirecting requests for mirrored data back to the capacity device. Cerberus matches HeMem's performance under low load and achieves 1.53× higher throughput during workload bursts since Cerberus also utilizes the capacity



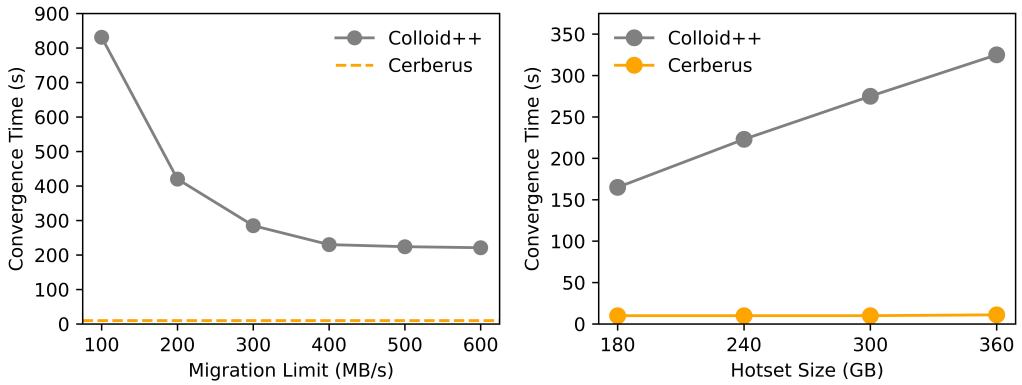
(a) Write-only **(b) RW-mixed**

Figure 4.6: Dynamic Bursty Workload: Write-only and RW-mixed. *Same setup as Figure 4.5. Colloid++ migrates 214 GB and 260 GB to the performance device and 186 GB and 239 GB to the capacity device under workloads (a) and (b), respectively. In comparison, Cerberus migrates 107 GB and 64 GB of data to the capacity device.*

device.

Write-only and Read-Write Workloads. The other workloads in Figure 4.6(a) and Figure 4.6(b) demonstrate similar behavior. Cerberus is able to load balance aligned 4 KB writes by using subpages that track the valid versus invalid portions of each 2 MB segment. For the write-only workload, Cerberus achieves 1.48× higher throughput than HeMem under high load conditions.

Disadvantages of Migrating Data. Colloid's reliance on migrating data has a number of disadvantages: not only does migration interfere with foreground traffic, but it also increases device writes and the time required for convergence. As shown in the captions of Figures 4.5 and 4.6, Colloid migrates on average 252 GB to the performance tier and 229 GB to the capacity tier across three representative workloads. In contrast, Cerberus only mirrors on average 86 GB of data to the capacity tier. For instance, under a read-only workload, Colloid issues 282 GB to the performance tier and 262 GB to the capacity tier, while Cerberus writes just 87 GB. Running this workload for



(a) Migration Limit **(b) Hotset Size**

Figure 4.7: Limitation of Migration-based Approach. *The same workload as Figure 4.5.*

one day results in 6.6 DWPD and 3.1 DWPD on the performance and capacity tiers, respectively. With a warranted endurance of 30 DWPD over 5 years [8], the performance-tier device lasts 5.0 years with Cerberus. Colloid reduces the lifespan to 4.1 years, an 18% drop. For the 1 TB capacity-tier device rated for 0.37 DWPD over 3 years [15], Colloid introduces 3.1 DWPD of migration writes, shortening the lifespan from 3.0 years to 129 days, an 88% reduction.

While Colloid's migration overhead can be controlled, limiting the rate of migration increases Colloid's convergence time, which is already significantly longer than that of Cerberus. Figure 4.7(a) shows Colloid's convergence time on the read-only workload when transitioning from low to high load given migration limits between 100 MB/s (5 DWPD) to 600 MB/s (30 DWPD). With a limit of 100 MB/s, Colloid requires more than 800 seconds to adapt to a load increase, making it ineffective for workloads containing shorter burst intervals. In contrast, Cerberus requires less than 10 seconds to adapt to load changes.

Relying exclusively on migration to adapt to workload changes also causes convergence time to increase as the size of the hotset increases. Figure 4.7(b) shows that Colloid's convergence time increases with a larger (read-only) hotset since more data must be demoted to offload traffic from

the performance device to the capacity device. In contrast, Cerberus's convergence time is independent of the hotset size, since no migration is performed once the data is mirrored.

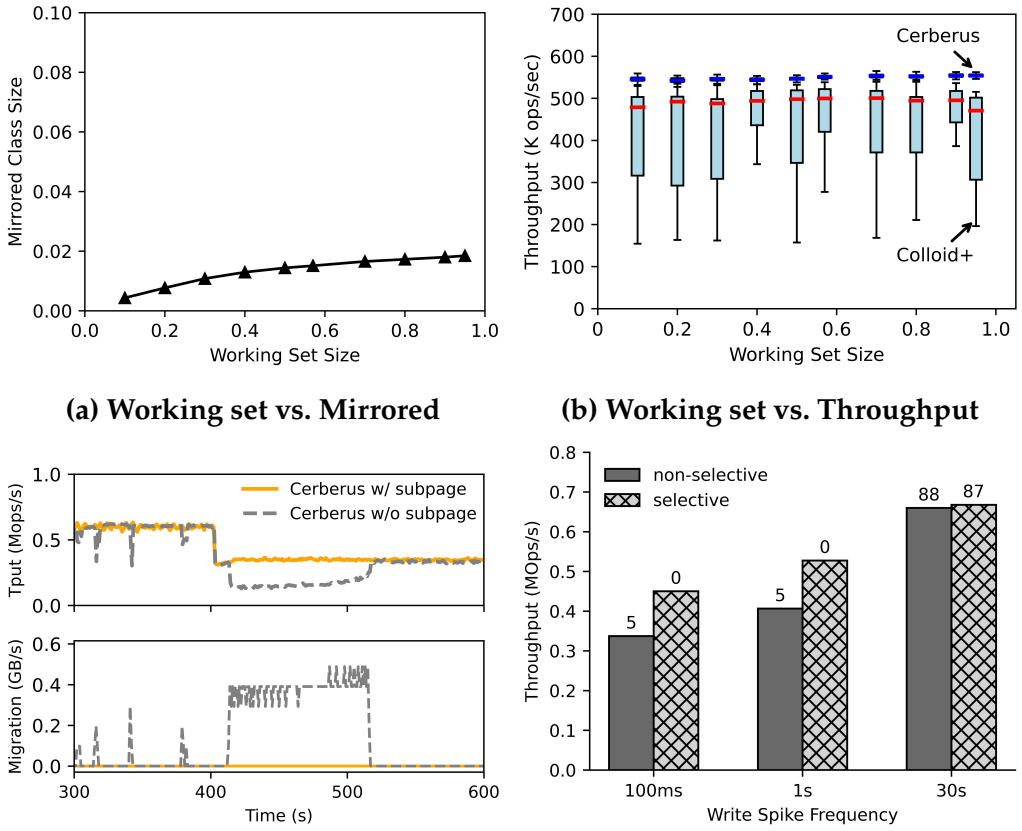
Summary. Cerberus integrates mirroring into classic tiering, allowing load adjustments without frequent migrations; this significantly enhances adaptability to bursty workloads and reduces overall write traffic. In contrast, Colloid and other tiering approaches lack intra-tier redundancy and rely exclusively on migration for load balancing, since data can only be accessed from one location; under bursty workloads, Colloid incurs extensive writes for migration, greatly reducing device lifespan and resulting in performance worse than HeMem.

4.3.3 Cerberus In-depth Analysis

We now perform an in-depth analysis of Cerberus's techniques for determining the size of the mirrored class, tracking the state of subpages in mirrored data, and selective cleaning.

Mirrored Class Size. Cerberus performs efficient load balancing with a relatively small mirrored class. Figure 4.8(a) shows the amount of space needed for Cerberus's mirrored class as a function of the workload's working set size. Even when the working set size reaches 95% of the total system capacity, Cerberus only mirrors 1.8% of the total data. Figure 4.8(b) shows the corresponding throughput of Colloid+ and Cerberus. The throughput of Colloid+ is highly unstable due to interference from frequent migrations; Cerberus consistently demonstrates higher and more stable throughput due to its effective use of a small amount of mirrored data.

Mirrored Data Subpages. The use of subpages within the mirrored class enables Cerberus to efficiently handle write requests. Figure 4.8(c) compares the behavior of Cerberus with subpages to Cerberus without subpages on a 4 KB write-only workload undergoing a sudden load drop (from 128 to 8 threads) at 400 seconds. After the load drop, Cerberus with subpages immediately redirects writes back to the performance device,



(a) Working set vs. Mirrored **(b) Working set vs. Throughput**

(c) Subpage Management **(d) Selective Cleaning**

Figure 4.8: In-depth Analysis. *In (a), workload is random read/write mixed (50% writes) at a high load (128 threads), varying the working set size, which is shown as a fraction of the total storage system capacity. The hotset is 20% of the working set with 90% access probability. In (d), the workload is a 256-thread workload with occasional write spikes every 0.1, 1, and 30 seconds.*

quickly adapting to light load without requiring any data migrations. In contrast, Cerberus without subpages must migrate entire segments back to the performance device, as each write request is smaller than the segment size (2 MB). Cerberus without subpages incurs additional migrations and significantly longer convergence times.

Selective Cleaning. Cleaning of mirrored data is rarely needed in Cerberus: cleaning is needed only for workloads containing a relatively small percent-

Name	Get	Set	LoneGet	LoneSet	Key Size (B)	Avg Value Size (B)
A (flat-kvcache)	0.98	0	0.02	0	16-255	335
B (graph-leader)	0.82	0	0.18	0	8-16	860
C (kvcache-reg)	0.87	0.12	1.04e-05	0.003	8-16	33112
D (kvcache-wc)	0.6	0	8.2e-06	0.21	8-16	92422

Table 4.4: Production Trace Distributions from CacheBench. *Flat-kvcache and graph-leader are from an application cache with small value sizes, resulting in mostly random traffic; kvcache-reg and kvcache-wc are from a storage cache with large value sizes, leading to log-structured traffic. LoneGet and LoneSet are requests for a key not present in the cache.*

age of writes that invalidate a mirrored copy of data that is then frequently read (e.g., caching for ML models where write spikes occur when model parameters are refreshed [129]). Figure 4.8(d) shows the efficiency of Cerberus's selective cleaning under a read-intensive workload with occasional write spikes. The number atop each bar represents the clean data percentage. We make two key observations. First, non-selective cleaning results in a 25% decrease in cache throughput, but only a 5% increase in the clean block percentage, because it wastes effort cleaning frequently written data. Second, Cerberus's selective cleaning policy effectively filters out data subject to high-frequency writing while cleaning data written at long intervals (every 30 s).

4.3.4 CacheLib

We now compare the end-to-end performance of CacheLib using Cerberus for its storage management layer, as compared to striping (CacheLib's default), caching (Orthus), and tiering (HeMem, Colloid, and Colloid++). For workloads, we use CacheBench [3], a highly configurable benchmarking tool bundled with CacheLib that can model real-world production cache workloads [4]. We also extend CacheBench to support Zipfian distributions and dynamic workloads.

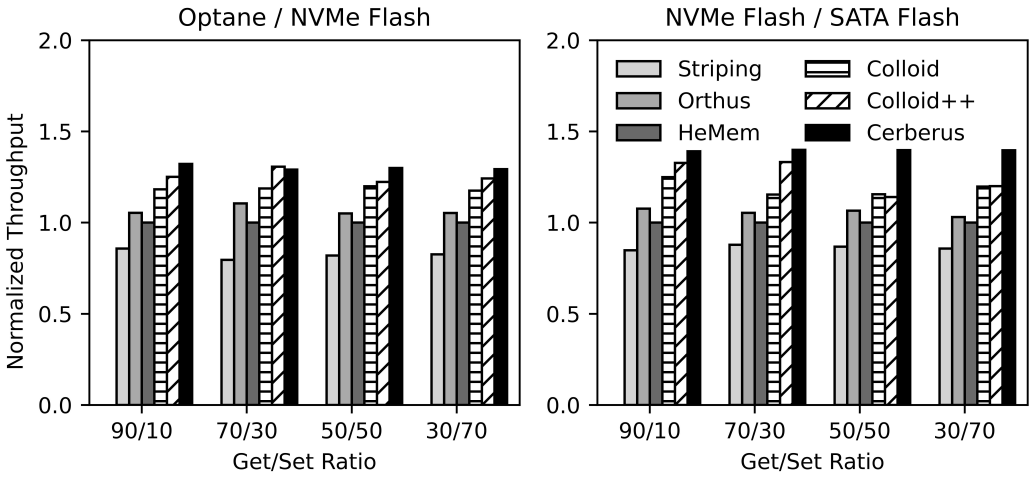


Figure 4.9: Lookaside Cache Workload: Small Object Cache. *The DRAM cache is restricted to 200 MB to stress the SOC in CacheLib. The workloads are 256 threads get/set-mixed Zipfian with 25M keys and 1 KB value. SOC is set to 100 GB, one third of the total capacity (100 GB/200 GB hierarchy).*

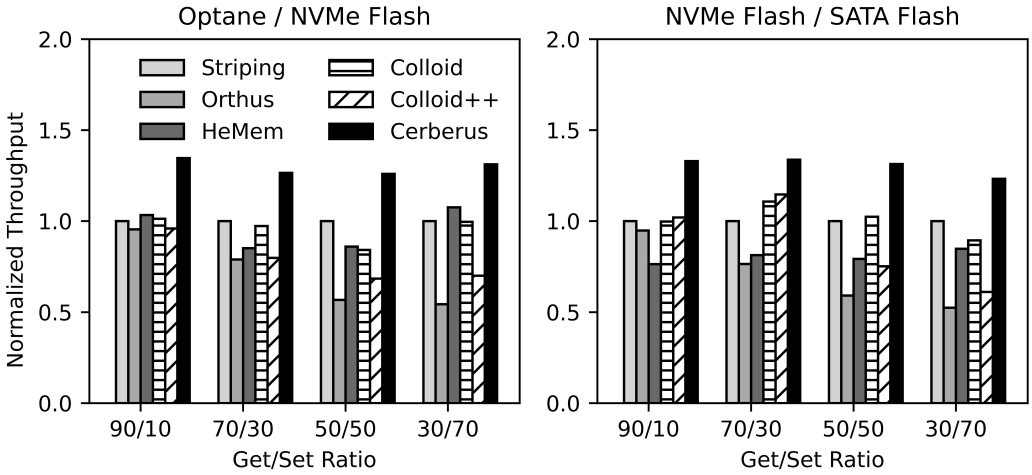


Figure 4.10: Lookaside Cache Workload: Large Object Cache. *Same setup as Figure 4.9, but the workloads are 256 threads get/set-mixed Zipfian with 5M keys and 16 KB value to stress the LOC.*

4.3.4.1 Static Workloads

We examine workloads that stress CacheLib's Small Object Cache (SOC) as well as its Large Object Cache (LOC).

Small Object Cache. We begin with 1 KB lookaside cache workloads varying the Get/Set ratio across the two storage hierarchies. Figure 4.9 shows that striping, Orthus, and HeMem continue to deliver suboptimal performance. Colloid and Colloid++ perform worse than Cerberus due to migration overhead caused by latency spikes, especially on the NVMe/SATA Flash capacity layer where NVMe and SATA exhibit more severe read/write interference than the Optane device.

Large Object Cache. The 16 KB lookaside cache workload stresses the Large Object Cache, which stores large key-value pairs in a sequential log and an in-memory index; this results in sequential writes to the storage management layer and reads primarily to the most-recently-written blocks (i.e., the log head). Figure 4.10 shows that HeMem and Colloid cannot utilize the capacity device bandwidth once the performance device becomes saturated. Cerberus performs optimally across all workloads, achieving up to 1.36× higher throughput on Optane/NVMe and up to 1.54× higher throughput on NVMe/SATA.

CPU Overhead. Under the 4 KB random workload that stresses the mirroring mechanism most (Figure 4.9), Cerberus slightly increases CPU utilization (0–1.5%) at 256 threads compared to the best-performing baseline, Colloid++, due to mirrored-page tracking and routing logic.

4.3.4.2 Production Cache Workloads

We evaluate four long-running production cache workloads provided by Meta [4], as detailed in Table 4.4. Figure 4.11 summarizes the results. For Workload A (lookaside cache workload) and Workload B (graph leader workload), Cerberus performs slightly better than Colloid and Colloid++, as these workloads involve small key-value pairs and stress the Small Object

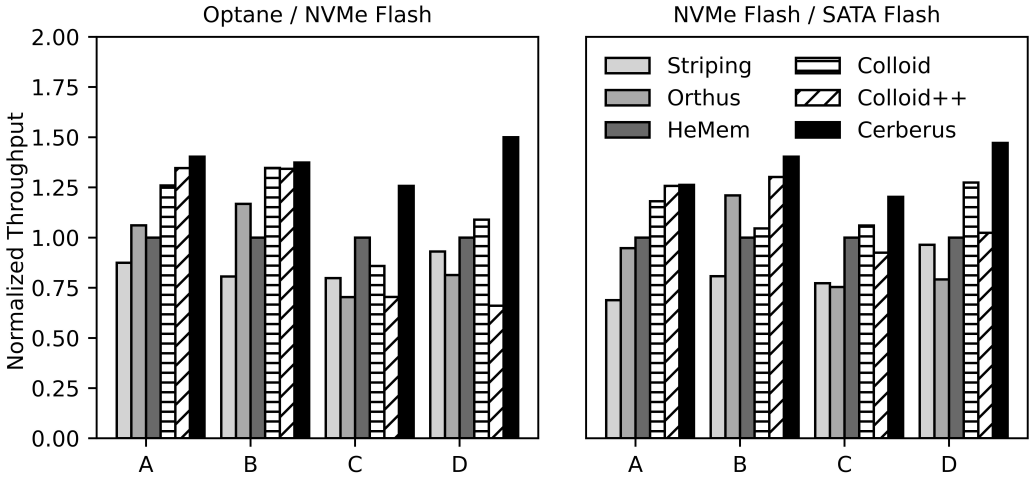


Figure 4.11: Production Workloads. *DRAM cache is set to 1 GB. For flat-kvcache and graph-leader, the SOC is set to one third of the total hierarchy capacity. We use 256 threads for flat-kvcache, graph-leader, and kvcache-wc and 80 threads for kvcache-reg. The cache throughput is normalized to HeMem's performance.*

Cache. On Workloads C and D, Cerberus significantly outperforms others, as these stress the Large Object Cache (LOC); Cerberus effectively distributes writes through its dynamic write allocation. Cerberus demonstrates better performance across all four production workloads due to its efficient mirroring-based load balancing and dynamic allocation. Compared to Colloid, Cerberus achieves an average throughput improvement of 1.24× on the Optane/NVMe hierarchy and 1.17× on the NVMe/SATA hierarchy.

Latency. Table 4.5 shows that Cerberus reduces average latency by 14% and P99 latency by 19% on average compared to the best-performing baseline. Striping performs worst on workloads A and B due to bottlenecks from the slower device, while Orthus performs worst on log-heavy workloads C and D, consistent with Figures 4.4(c) and 4.4(d). Performance gains are more substantial on the Optane/NVMe hierarchy, with 20% lower average latency and 26% lower P99 latency compared to the best-performing baseline. In comparison, the NVMe/SATA hierarchy shows smaller improvements, with 6.6% and 12% reductions respectively.

Device	Workload	Metric	Striping	Orthus	HeMem	Colloid	Colloid++	Cerberus
Optane NVMe	A	Avg (ms)	0.81	0.67	0.71	0.56	0.52	**0.50**
		P99 (ms)	15.76	18.18	19.24	9.59	9.00	**7.33**
	B	Avg (ms)	0.73	0.61	0.59	0.45	0.44	**0.43**
		P99 (ms)	7.00	14.28	12.83	4.12	4.12	**3.65**
	C	Avg (ms)	0.76	0.86	0.59	0.67	0.88	**0.47**
		P99 (ms)	9.53	10.69	6.50	7.78	8.85	**5.43**
	D	Avg (ms)	9.67	12.38	8.26	7.15	8.32	**3.16**
		P99 (ms)	67.68	272.00	73.63	72.93	86.32	**27.76**
NVMe SATA	A	Avg (ms)	2.30	1.76	1.58	1.35	1.21	**1.15**
		P99 (ms)	46.43	27.96	22.98	25.98	21.31	**20.40**
	B	Avg (ms)	1.90	1.41	1.54	1.48	1.17	**1.10**
		P99 (ms)	23.40	16.84	11.59	14.15	9.34	**8.54**
	C	Avg (ms)	1.95	2.00	1.49	1.40	1.62	**1.23**
		P99 (ms)	18.18	18.42	12.35	16.50	18.50	**12.31**
	D	Avg (ms)	27.55	32.15	26.69	19.98	16.68	**16.09**
		P99 (ms)	239.86	364.20	160.10	158.07	170.29	**101.85**

Table 4.5: Average and P99 GET Latency (ms) of Production Workloads.

4.3.4.3 Dynamic Workload

We compare how well Colloid and Cerberus handle load changes using the CacheBench benchmark [4]. The bursts happen every 180 seconds and last for 60 seconds, matching those studied in [23] for data center workloads. Figure 4.12 shows that Colloid struggles to adapt to these bursty workloads, generating significant migration traffic. In contrast, Cerberus efficiently adapts to bursty workloads without incurring any migration overhead.

4.3.4.4 YCSB

We compare Cerberus to striping, tiering (HeMem), and caching (Orthus) under the YCSB benchmark across two local hierarchies shown in Figure 4.13. Since YCSB does not natively handle cache misses, we extend it to implement a lookaside caching pattern [23], where cache misses trigger a fetch from the backing store (simulated with a 1.5 ms delay) and re-insertion into the cache.

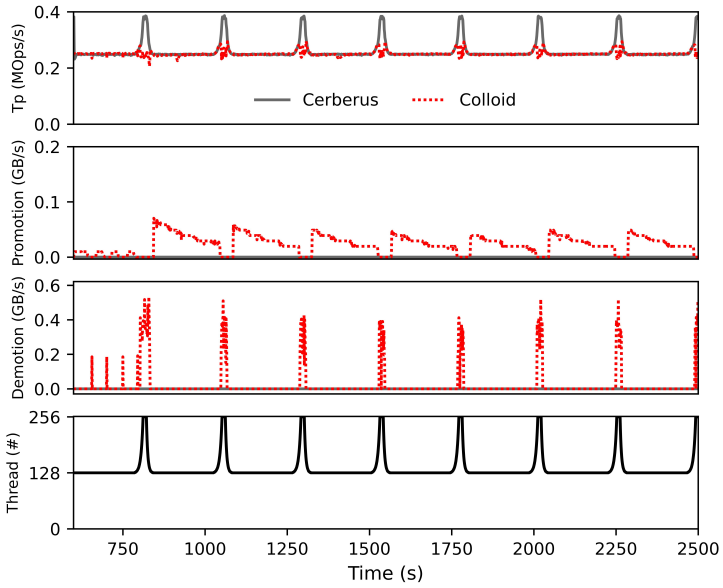


Figure 4.12: Dynamic Cache Workload. *Read-heavy workload with 95% GET and 5% SET operations on Optane/NVMe hierarchy, where 20% of keys belong to the hotset; the hotset is accessed uniformly at random with a 90% probability. 25 million key-value pairs, with value sizes ranging from 2 KB to 4 KB; 1 billion operations. Size of SOC is 450 GB.*

Cerberus shows up to 1.43× higher throughput and 30% lower P99 latency compared to the best-performing system.

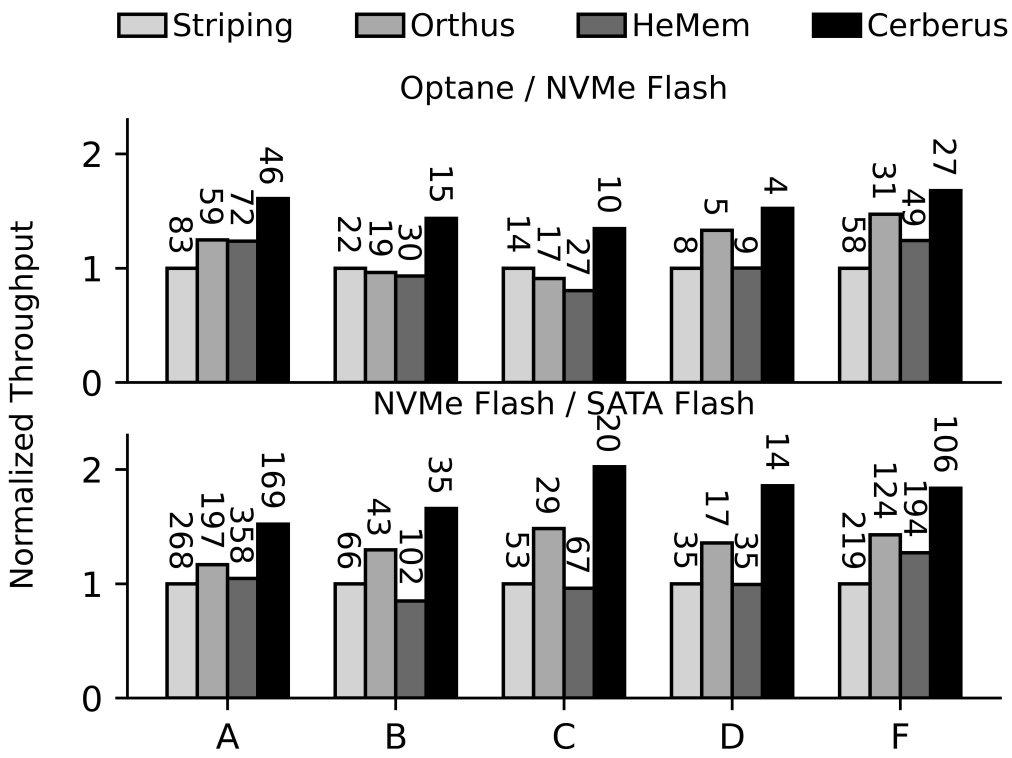


Figure 4.13: YCSB. *CacheLib uses the default parameters from CacheBench with 4 GB DRAM cache. All the workloads are Zipfian ($\theta = 0.8$) with 1 KB values and 16-byte keys running under 256 threads with 20M records. The cache throughput is normalized to the throughput of the default (striping) system and the number above each bar is P99 latency in microseconds. Workload E is excluded due to CacheLib's lack of support for range queries.*

4.4 Discussion

Multi-tier Extensions. MOST's mechanism naturally extends beyond the two-tier scenario. For example, data can be mirrored across multiple tiers, allowing requests to be dynamically routed to the tier with the lowest latency. This generalization calls for a more sophisticated optimization policy, which we leave as future work.

Consistency. MOST can be extended to provide stronger consistency guarantees. One possible approach is to maintain a write-ahead log for mapping

updates, such as those triggered by data migration. We leave a full study as future work.

Performance Isolation. Currently, MOST transparently manages the underlying storage devices at the block level and is unaware of which tenant a given request belongs to. One potential solution is to use hints to associate each request with its corresponding tenant. With this additional metadata, MOST can be extended to support and enforce performance isolation policies, such as fairness and quality of service (QoS), across multiple tenants.

4.5 Conclusion

We introduced Mirror-Optimized Storage Tiering, a new tier-based approach optimized for modern heterogeneous storage hierarchies. We demonstrate that with a small amount of mirrored data, MOST improves throughput, reduces latency, enhances adaptability, and decreases migration-induced writes under dynamic workloads. In the future, hybrid approaches that integrate elements of caching, tiering, and RAID may offer further benefits, potentially leading toward a "unified theory" of storage hierarchy management.

Chapter 5

Mapping-Based Balancing in Non-Hierarchical Memory

Chapters 3 and 4 addressed non-hierarchical storage systems by rethinking caching and tiering. The same phenomenon arises in non-hierarchical memory systems, where local DRAM is paired with second-level memory such as CXL-attached memory: under interconnect contention, the loaded latency of the performance tier can rise above that of the capacity tier. In this chapter, we focus on load balancing in heterogeneous memory and propose Hecaton, a tiered-memory system that introduces mapping-based load balancing as an alternative to migration-based balancing. This chapter is based on the paper *Hecaton: Mapping-Based Load Balancing for Contended Heterogeneous Memory*.

In the following, we first motivate mapping-based load balancing against migration-based contention-aware systems (§5.1). We then present the design of Hecaton (§5.2) and evaluate it against HeMem and three Colloid configurations on dynamic workloads, bursty workloads, and realistic memory-intensive applications (§5.3). Finally, we conclude (§5.4).

5.1 Motivation and Background

This section motivates low-cost load balancing in heterogeneous memory. We first review how contention-oblivious systems place data across tiers, implicitly assuming that the performance tier is always faster. We then describe contention-aware systems, which react when memory-interconnect contention undermines this hierarchy, and show why migration-based balancing can be too slow and disruptive for bursty contention. Finally, we discuss how read-mostly workloads and mirrored data create an opportunity to balance load through page table remapping without data copying under contention.

5.1.1 Load Balancing in Heterogeneous Memory

A heterogeneous memory system manages multiple memory technologies as a single large memory transparently to the application. These systems usually pair a fast, smaller local DRAM tier with a slower, larger capacity tier, such as CXL devices [35, 66, 85, 103], or persistent memory [96]. We use *performance tier* and *capacity tier* to describe the intended roles of these tiers, even though their loaded latencies can change at runtime. These heterogeneous memory systems are non-hierarchical [123]: under unloaded conditions, capacity devices have only 2–3× higher latency than local DRAM [67]. Therefore, under contention, the performance tier's loaded latency can exceed the capacity tier's [19, 54, 116, 117], which we call *latency inversion*, breaking the usual assumption that the performance tier is always faster.

Existing systems respond to latency inversion in two broad ways. *Contention-oblivious systems* do not react and therefore perform suboptimally under contention [18, 56, 58, 64, 68, 68, 77, 96, 98, 107, 126–128, 138]. For example, HeMem [96] addresses the high overhead of memory access tracking by using PEBS to efficiently identify hot pages for promotion to the performance tier. Nomad [126] retains the capacity-tier copy during and after page promotion to mitigate memory thrashing. However, both systems place hot data on the performance tier and serve the majority of accesses from there. Therefore,

they fail to utilize the capacity tier when the performance tier is contended.

Contention-aware systems react to contention by adjusting load distribution across the two tiers so that both can be utilized. These systems can be categorized at a high level by their load-balancing mechanism: migration-based and routing-based. *Migration-based load balancing* (Figure 5.1a) keeps a single copy of each page and physically moves data across tiers when load shifts. For example, Chou et al. propose BATMAN [32], which shrinks the performance tier by demoting pages until the access ratio between the two tiers matches a predefined bandwidth ratio. Vuppalapati et al. propose Colloid [116], a system with a dynamic load balancer that equalizes the loaded access latency of the two tiers instead of using a predefined threshold. *Routing-based load balancing* (Figure 5.1b) maintains two copies and routes each request to one of them proportionally. For example, Tu et al. propose MOST [114] for storage tiering, which mirrors a subset of data across tiers and routes each request to either tier to adjust the load distribution.

Both approaches have limitations in heterogeneous memory systems. Routing-based load balancing is not directly applicable because memory accesses are issued by CPU instructions and resolved by hardware through page table lookups, leaving no software layer that can intercept and route individual accesses. Migration-based load balancing is applicable, but requires copying data under contention. This slows convergence and interferes with the foreground workload by adding traffic to an already contended memory subsystem, as shown in prior work [68, 116, 126]. To balance load without significantly interfering with foreground traffic, a migration-based system must limit its migration rate and may converge over tens of seconds to minutes [116].

Table 5.1 summarizes this design space. Existing contention-oblivious systems optimize mechanisms for access tracking, page migration, dynamic page size determination, and thrashing mitigation, but do not balance load when contention changes the relative latency of the two tiers. For example, Nomad retains a capacity-tier copy to mitigate memory thrashing under memory pressure, but does not use the retained copy for load balancing.

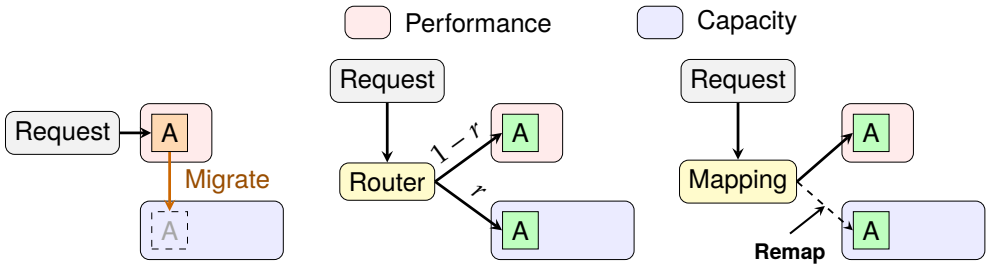


Figure 5.1: Three approaches to load balancing. *Migration-based balancing keeps a single copy and moves data when load shifts; it is applicable to both memory and storage systems. Routing-based balancing maintains two copies and routes each request to one of them; it is not applicable to memory systems. Mapping-based balancing maintains two copies and switches the page table mapping to redirect future accesses.*

Existing contention-aware memory systems do rebalance load, but use migration as the balancing mechanism. Hecaton fills the remaining design point: contention-aware load balancing that balances load through page remapping without copying data under contention.

5.1.2 Why Migration-Based Balancing is Insufficient

Real-world workloads often change their memory-bandwidth demand on timescales of only a few seconds [41, 53]. Graph analytics workloads [21], for example, keep a graph resident in memory and trigger short analytics jobs on demand: in our experiment, PageRank on a web graph runs for around 5 seconds, while BFS runs for 1 to 5 seconds depending on concurrency and graph structure. These jobs can consume a large fraction of available memory bandwidth (up to 90% in our measurement). Contention can also come from peripheral devices such as NICs and storage [117]; datacenter network traffic and scientific I/O are both highly bursty, often lasting only a few seconds [71, 75]. Such short-lived contention is a poor match for migration-based balancing: the system must shift load before the burst ends, but copying pages adds traffic to the same memory subsystem that is already contended.

Contention Response	Systems	Data Layout	Placement	Load Balancing		
				Mechanism	Convergence	Interference
Contention oblivious	HeMem [96], Memtis [64], HybridTier [107]	Single copy	Hotness	–	–	–
	SoarAlto [68], PACT [47]	Single copy	Perf. criticality	–	–	–
	Nomad [126]	Multi copy	Hotness	–	–	–
Contention aware	BATMAN [32]	Single copy	Hotness	Migration	Slow	High
	Colloid [116]	Single copy	Underlying	Migration	Slow	High
	Hecaton	Multi copy	Underlying	Mapping + Migration	Fast	Low

Table 5.1: Comparison of Heterogeneous Memory Systems. *Data Layout specifies whether the system maintains one or multiple copies of each page. Placement refers to the criterion the system uses to decide where to place data. For convergence and interference, Fast/Low are desirable, Slow/High are undesirable, and – indicates no load balancing.*

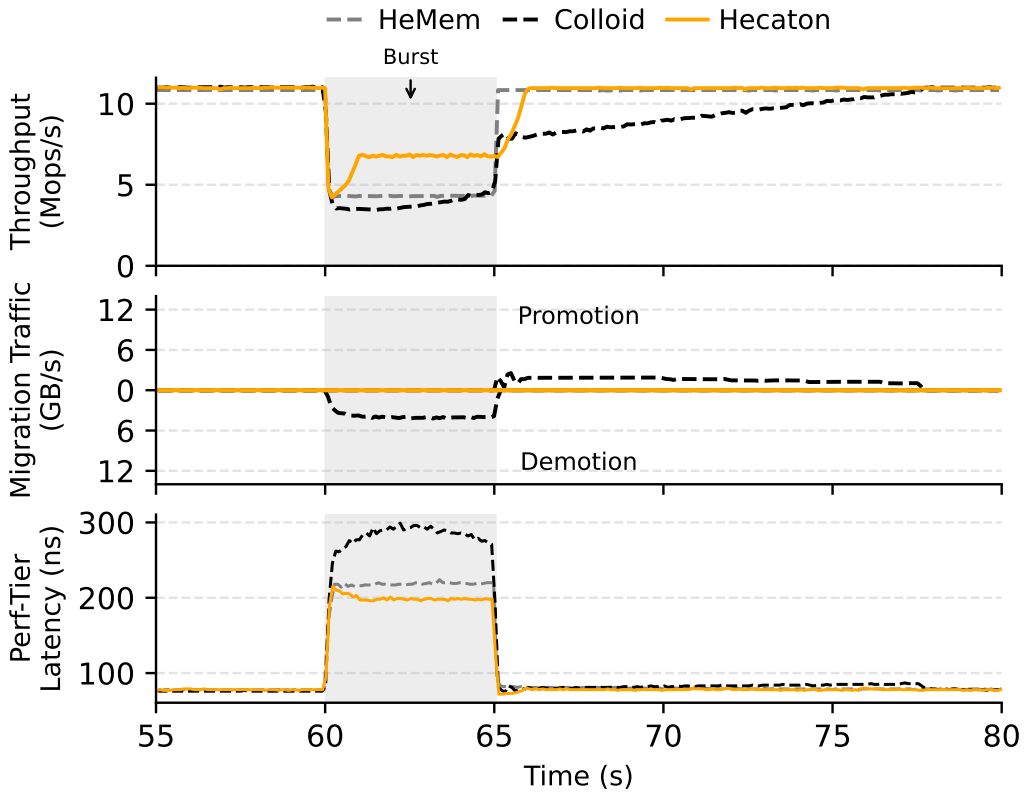


Figure 5.2: Load Burst. *A 4-thread multichase workload under a 5-second burst from an 8-thread antagonist, which raises performance-tier latency to 1.5× that of the capacity tier (grey region).*

To understand how existing systems behave under bursty contention, we run a multichase workload [45] whose 32 GB working set is initially placed in the performance tier. In the middle of the run, we co-run an antagonist workload that creates a 5-second burst of contention on the performance tier. We compare HeMem, a contention-oblivious system, with Colloid, a state-of-the-art migration-based contention-aware system. Figure 5.2 shows throughput, migration traffic, and performance-tier latency over time. We include Hecaton as a preview of the opportunity enabled by mapping-based balancing; Section 5.3 evaluates it in detail. We make four key observations from this experiment.

Contention-oblivious systems perform suboptimally under contention.

As shown in Figure 5.2, HeMem does not react when the performance tier becomes contended. It therefore continues to serve the workload from the performance tier, even though the capacity tier has lower loaded latency during the burst. A contention-aware system can improve throughput by shifting load to the capacity tier; for example, Hecaton achieves 1.45× higher throughput than HeMem during the burst.

Migration-based balancing may not converge before the load changes. Colloid has the right high-level goal: it attempts to balance load during the burst by demoting pages to the capacity tier. However, because each demotion requires copying page contents, the burst ends before Colloid converges. When the burst ends, the demoted pages are now misplaced and must be promoted back to the performance tier. During this recovery period, Colloid performs worse than HeMem because some pages remain on the slower capacity tier until they are migrated back. As a result, Colloid delivers on average 11.4% and 15.1% lower throughput during burst and recovery, respectively.

Migration interferes with the foreground workload. Colloid also causes significant foreground interference during convergence. As shown in the bottom graph of Figure 5.2, Colloid raises performance-tier latency by 1.28× on average. This interference occurs because migration injects additional traffic into the already-contended performance tier, increasing request queuing. Increasing the migration rate would reduce convergence time but further increase interference; reducing interference requires lowering the migration rate, which makes the system even less able to adapt to short bursts.

Migration causes repeated data movement. The middle graph in Figure 5.2 shows that, to react to a single load burst, Colloid migrates 48 GB of data: 20 GB of demotions during the burst and 28 GB of promotions during recovery. This cost is paid on every burst. When contention occurs periodically, migration-based systems can generate excessive background traffic and affect the lifetime of write-limited devices such as CXL-attached SSDs [130].

The root cause is that, with a single active copy of each page, changing the load distribution requires moving data. If pages had useful copies in

both tiers, the system could instead redirect future accesses without copying page contents. As shown in Figure 5.1, routing-based balancing routes each request to one of two copies through a load balancer, but this approach is not directly applicable to memory systems because memory accesses are resolved by hardware page table lookups, leaving no software layer to intercept individual accesses. An alternative is to use existing virtual-memory hardware to redirect future accesses by changing page table mappings. We call this approach mapping-based load balancing (Figure 5.1c). We next describe why memory-intensive workloads create an opportunity for this mechanism.

5.1.3 Opportunity: Mapping-Based Load Balancing

Migration-based balancing is limited by its single-copy layout: shifting load requires moving page contents. If useful copies already exist in both tiers, a system can instead redirect accesses without copying data. Storage systems can do this by routing individual requests to different copies, but memory systems do not have software between CPU loads/stores and the data. Instead, the available control point is the page table. We call this approach *mapping-based load balancing*: the system maintains copies of selected pages in both tiers and shifts future accesses by changing page table mappings.

Mapping-based load balancing is a good fit for memory-intensive workloads because they are often dominated by read-intensive data. Many workloads, including graph analytics, tree indexes, ML, and HPC, preallocate large objects such as graphs, indexes, embedding tables, and training datasets that are read-only after initialization [64, 96, 110, 116]. To quantify this opportunity, we profile representative memory-intensive workloads used in tiered-memory evaluation, including database indexes [64, 115], ML kernels [39, 81, 110], graph analytics [21], and HPC benchmarks [113]. These workloads span a range of working-set sizes, access skews, memory-level parallelism, and access patterns, allowing us to test whether the read-mostly property holds beyond a single application style. We build a tool on Intel

PIN [97] that records per-page load/store patterns. As shown in Table 5.2, over 95% of each working set is read-only. This implies that, for these memory-intensive workloads, most memory can be mirrored and balanced through mapping-based load balancing.

This observation does not require every page to be read-only or every page to have a mirror. Frequently written pages can be handled through conventional migration, and when no spare capacity remains for mirroring, the system can fall back to migration-based balancing.

5.2 Hecaton Design

We propose Hecaton, a heterogeneous memory management system that introduces mapping-based load balancing as a fast and low-cost alternative to migration-based load balancing. The key idea is to eliminate expensive data copying under contention by maintaining mirrored copies of hot, read-intensive pages across tiers and balancing load through page table remapping. When clean mirrors are unavailable, Hecaton falls back to migration-based balancing, allowing it to degrade gracefully to a conventional contention-aware system. Realizing this idea in a tiered memory system raises three challenges:

- How can the system balance load fast enough to react to second-scale load changes? (§5.2.2)
- How can the system create and maintain mirrored copies with minimal overhead, while safely handling writes that make mirrors stale? (§5.2.3)
- How can the system accurately identify read-mostly pages worth mirroring? (§5.2.4)

Hecaton requires no prior knowledge of the workload or hardware. We begin with an overview of the basic design before addressing each challenge.

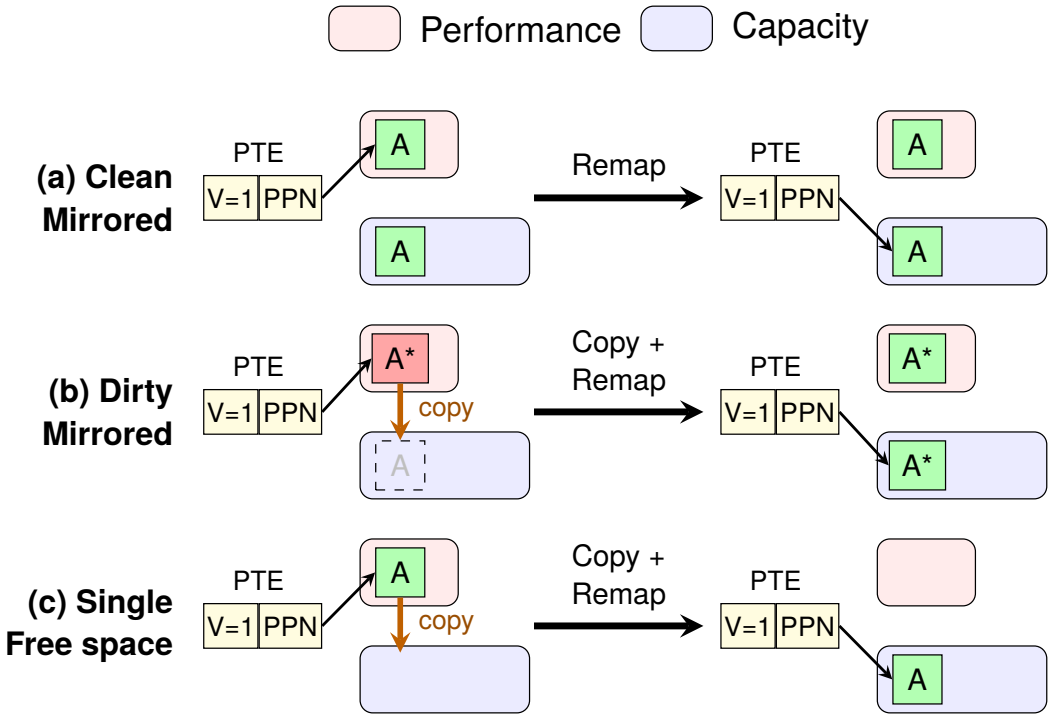


Figure 5.3: Hecaton Balancing Paths. *(a) Clean mirrored: remap only. (b) Dirty mirrored: overwrite the stale copy, then remap. (c) Single-copy into free space: write to an empty slot, then remap.*

5.2.1 Overview

Hecaton extends a conventional tiered memory system by allowing selected pages to maintain mirrored copies on both tiers. As shown in Figure 5.3, each page is in one of three states: *clean mirrored* (both copies are consistent), *dirty mirrored* (the active copy has been written since the last mirror), or *single-copy* (no mirror exists). At any time, only one copy of a mirrored page is active: the copy named by the page table mapping.

Hecaton operates in two complementary modes. During idle periods, it selectively mirrors hot, read-intensive pages from the performance tier to the capacity tier. During contention, it balances load using the cheapest available action: remapping clean mirrored pages first, repairing or migrating dirty mirrored pages when needed, and migrating single-copy pages as a fallback.

Algorithm 3: Load Balancer

```
1  Function LoadBalance():
2      (L_p, L_c) ← measure latency of each tier
3      if L_p < (1 − θ) L_c then
4          Rebalance(perf)
5      else if L_p > (1 + θ) L_c then
6          Rebalance(cap)
7      else
8          No Balancing
9      end
10
11 Function Rebalance(tier):
12     l ← 0; m ← 0
13     foreach clean mirrored page p do
14         if l ≥ L_max then return
15         remap p to tier
16         l ← l + load(p);
17     end
18     foreach dirty mirrored page p do
19         if l ≥ L_max or m ≥ M_max then return
20         migrate p to tier
21         l ← l + load(p); m ← m + |p|;
22     end
23     foreach single-copy page p do
24         if l ≥ L_max or m ≥ M_max then return
25         swap p to tier
26         l ← l + load(p); m ← m + |p|;
27     end
```

5.2.2 Fast Load Balancing

Given mirrored pages, Hecaton must decide when to shift load, how much load to shift, and which pages can be shifted without copying data. Hecaton uses a lightweight feedback controller that runs once per epoch and compares the measured latency of the performance tier (L_p) and capacity tier (L_c). If $L_p < (1 - \theta)L_c$, Hecaton shifts load back to the performance tier; if $L_p > (1+\theta)L_c$, it shifts load to the capacity tier; otherwise, it leaves the system

unchanged. Algorithm 3 summarizes the controller and the rebalancing procedure.

The Hecaton controller must be lightweight because mapping-based load balancing makes page movement much cheaper than in migration-based systems. We measure that, in Colloid [116], controller logic takes 200 μs per page, comparable to the 264 μs spent moving data. In Hecaton, remapping a page takes only 49 μs, so Colloid's controller would become the bottleneck. Hecaton therefore avoids expensive per-page selection and binary search, reducing controller overhead by 8× to 24 μs per page.

Once the controller decides to rebalance, Hecaton selects pages for rebalancing in order of their cost, as shown in Figure 5.3.

① **Page Remapping (fast path).** For a clean mirrored page, Hecaton balances load by updating the page table entry to point to the copy on the target tier. This is the core mechanism of mapping-based load balancing and requires no data movement. Its cost is one remap: a PTE update and TLB shootdown.

② **Mirror Repair and Remapping.** For a dirty mirrored page, the active copy has been written and the inactive copy is stale. Hecaton copies the up-to-date version over the stale copy on the target tier and then updates the mapping. This action both restores consistency and shifts future accesses to the target tier. Because space for the copy already exists on both tiers, no swapping is needed. Its cost is one remap, one read, and one write.

③ **Promotion/Demotion.** For a page without a mirror, Hecaton falls back to standard migration-based balancing by promoting or demoting the page to the target tier. If the target tier has free space, this costs one read, one write, and one remap. When the target tier is full, Hecaton swaps out another page on the target tier to make room, increasing the cost to two reads, two writes, and two remaps.

Paths ② and ③ both generate background traffic, so Hecaton bounds their use with the per-epoch migration limit configured by the underlying system. Path ① generates no data movement, but Hecaton still bounds remapping with a per-epoch load step size to avoid overshooting the target load distribution. In each epoch, Hecaton applies paths in increasing order of

cost until it has shifted the target load step (2%) or exhausted the migration budget.

This three-path design lets Hecaton react quickly through the fast path while degrading gracefully to migration-based balancing when needed. When all pages are clean mirrored, load balancing is entirely through remapping. When pages are dirty or unmirrored, Hecaton falls back to migration for those pages while still remapping the clean mirrored pages.

5.2.3 Mirror Maintenance

Mapping-based load balancing requires mirrored copies, but maintaining them poses two challenges. First, creating a mirror requires copying data across tiers, which consumes memory bandwidth. Hecaton must hide this cost to avoid interfering with the foreground workload, especially during contention. Second, writes to a mirrored page make the inactive copy stale, turning a clean mirrored page into a dirty one. Hecaton must therefore track writes efficiently without requiring eager coherence between the two copies. Hecaton addresses these challenges with reactive and proactive mirroring, dirty-bit write tracking with lazy repair, and capacity-aware reclaim.

Reactive and proactive mirroring. Hecaton creates mirrored copies through two complementary mechanisms: *reactive mirroring* and *proactive mirroring*. With reactive mirroring, whenever a single-copy page is promoted or demoted, Hecaton retains the original copy on the source tier, so the page becomes mirrored at no additional copy cost. With proactive mirroring, Hecaton copies pages from the performance tier to the capacity tier during idle periods. Proactive mirroring runs only when two conditions hold: the capacity tier has spare capacity above the low watermark, and the current performance-tier latency is lower than that of the capacity tier. If either condition fails, proactive mirroring is skipped for that epoch, ensuring it does not compete with the foreground workload for bandwidth or capacity. Among eligible pages (§5.2.4), Hecaton mirrors the most heavily loaded pages first, maximizing the load that can later be balanced through the fast

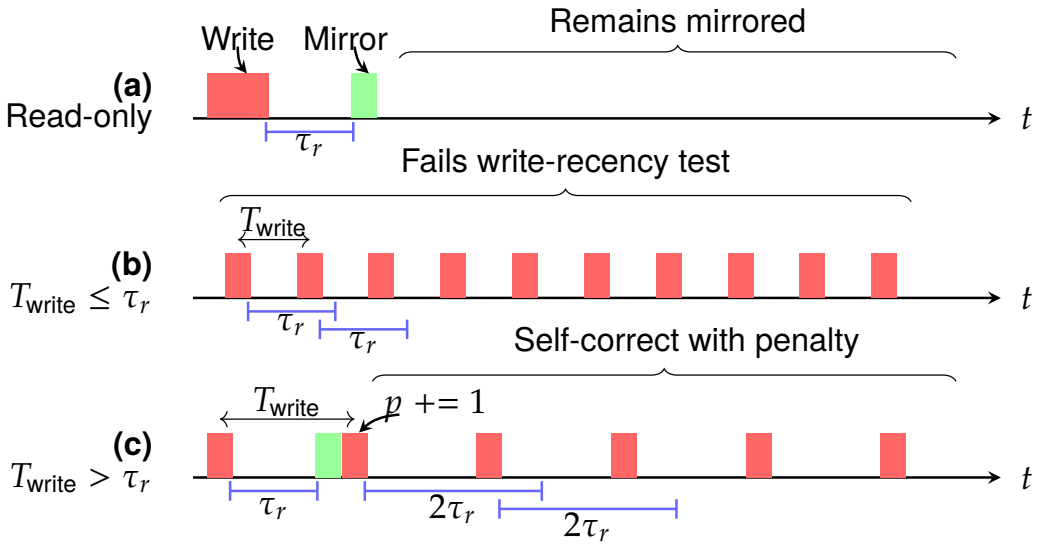


Figure 5.4: Read Set Identification. *Three cases: (a) read-only, (b) $T_{write} \leq \tau_r$, (c) $T_{write} > \tau_r$ with exponential penalty.*

path.

Hecaton's mirrors differ from retained copies in Nomad [126] in both purpose and maintenance. Nomad retains a capacity-tier copy after promotion to mitigate thrashing under memory pressure. In contrast, Hecaton maintains mirrors so that clean mirrored pages can be used for fast load balancing under memory-interconnect contention. This difference in purpose leads to different maintenance policies. Nomad creates retained copies only as a byproduct of promotion, while Hecaton proactively and selectively creates mirrors for pages on the performance tier. Nomad also write-protects the retained copy and discards it on a write, while Hecaton tracks writes lazily, allowing pages written during initialization to become useful mirrors later.

Handling writes with lazy repair. Writes to a mirrored page break consistency between the two copies, but Hecaton does not need to repair the inactive copy immediately. At any time, only one copy is active: the copy named by the page table mapping. Writes update this active copy and make the inactive copy stale. Rather than synchronously writing through to the inactive copy or trapping on every write as in previous systems [26, 126], Hecaton tracks writes using the hardware dirty bit. Whenever a page becomes

clean mirrored, Hecaton clears its dirty bit; if the page is later written, the MMU sets the dirty bit, and Hecaton marks the page as dirty mirrored. The stale copy is repaired only when doing so is useful: when spare bandwidth is available for re-mirroring or when the page must be moved to the tier holding the stale copy. This lazy repair avoids foreground write overhead while preserving correctness, because Hecaton never remaps to a stale copy without first repairing it.

Capacity-aware page reclaim. Because mirroring consumes capacity that a conventional tiered system would not use, Hecaton must reclaim mirrored pages when space pressure arises to prevent out-of-memory conditions. Reclaim is triggered when capacity-tier free space drops below the low watermark, the same threshold used by the Linux kernel [112]. Hecaton reclaims pages in the reverse order of mirroring priority, evicting the least-hot copies first. This keeps mirrored capacity focused on the hottest pages, maximizing the load that can be rebalanced through the fast path. For a clean mirrored page, the inactive copy is simply dropped. For a dirty mirrored page, the stale inactive copy is dropped while the active copy remains authoritative. In both cases, reclamation requires no data movement. When all mirrors are reclaimed, Hecaton falls back to conventional migration-based tiering with no performance penalty beyond losing the fast-path opportunity.

5.2.4 Identifying Useful Mirrors

Not all pages benefit equally from mirroring. A frequently written page quickly makes its inactive copy stale, wasting both mirror capacity and the bandwidth spent creating the mirror. Hecaton must therefore identify pages whose mirrors are likely to remain clean long enough to be useful for mapping-based load balancing.

Hecaton relies on two observations from the memory-intensive workloads we evaluate. First, most of the working set is read-intensive: pages are either never written or written a few times during initialization, accounting for over 95% of the working set across the workloads we profiled (§5.1). Second,

the remaining pages, such as stacks and mutable data structures, tend to be written frequently or periodically. These patterns make write recency a useful signal: a page that has remained unwritten for long enough is likely to be read-mostly, while a page that repeatedly dirties its mirror should become harder to mirror again.

Based on these observations, Hecaton uses a lightweight write-recency test with a self-correcting penalty for pages that dirty their mirrors. For page i, let Δ_i denote the time since its last write (or since allocation if never written), and let τ_r denote the read-set threshold. Hecaton classifies page i as a mirroring candidate only if $\Delta_i > \tau_r \cdot 2^{p_i}$, where p_i is a penalty counter that tracks how often page i has been dirtied after mirroring. Each time a freshly mirrored page becomes dirty, p_i is incremented once; subsequent writes to the same dirty copy do not further increase p_i until the page is mirrored again. Thus, a page must remain unwritten for a threshold interval before mirroring, and each failed mirroring attempt doubles the clean interval required before the page is considered again.

Figure 5.4 illustrates how the test keeps useful mirrors and filters out unstable ones. Initialization-only pages eventually pass the threshold and remain mirrored, while frequently written pages never stay clean long enough to pass. Periodically written pages may have write intervals longer than the threshold, but the penalty mechanism allows the algorithm to self-correct: when such a page dirties its mirror, the penalty doubles the required clean interval, preventing repeated mirroring of pages likely to become stale frequently.

5.2.5 Implementation

Base system. We implement Hecaton on top of Colloid [116], which itself is implemented on HeMem [96]. HeMem is a user-level, per-application tiered memory manager that provides PEBS-based hotness tracking, 2 MB page management, and an existing policy thread that runs every 10 ms. Colloid adds contention-aware, migration-based load balancing on top of

HeMem. Hecaton reuses this stack and adds mapping-based balancing as a fast path for clean mirrored pages, while retaining Colloid's migration mechanism as the fallback for dirty mirrored and single-copy pages. This structure lets Hecaton compare directly against the strongest migration-based implementation available. Because Hecaton inherits HeMem's 2 MB page granularity, all hotness tracking, mirroring decisions, dirty tracking, remapping, and migration operate on 2 MB pages. This granularity provides a fair comparison with HeMem and Colloid and keeps metadata, page table updates, TLB shootdowns, and controller overhead low. The tradeoff is precision: a write to any location in a 2 MB page makes that page's mirror dirty, even if most of the page is read-only. For example, in the BFS workload we observe 2.9× write-set amplification when managing pages at 2 MB granularity compared to 4 KB. Smaller pages could reduce this false dirtying and provide finer-grained load balancing, but would increase metadata and remapping overhead. We leave a full exploration of page-size tradeoffs and mixed page sizes to future work.

On-line latency estimation. Previous systems measure per-tier latency using hardware performance counters [116], which we find are inaccurate under low contention, insensitive to write traffic, and vendor-specific. We instead use an online latency-estimation approach: once per epoch, a 1 ms pointer-chase is run on each tier over a buffer sized to the aggregate CPU cache to avoid cache effects. We empirically found 1 ms to be the minimum for stable measurements. The probe uses negligible capacity and bandwidth (<0.5% each) and runs on HeMem's existing policy thread, so it requires no additional core. The latency signal is smoothed using EWMA, following prior work [114, 116].

Parameters. Hecaton exposes a small set of control parameters. The load-balancing step size is 2% of the load, and the latency tolerance θ is 5% as in other systems [114, 123]. Hecaton's read-set identification is insensitive to τ_r because the self-correcting penalty fixes misclassifications at runtime when a frequently written page has a write interval longer than the threshold. We set τ_r to 500 ms based on profiling: write-set pages in our workloads have

write intervals shorter than this threshold.

Metadata. Hecaton adds 32 bytes of metadata per 2 MB page: the address of the mirror, the last write timestamp, the penalty counter for read-set identification, and a dirty bit. For a system with a 32 GB performance tier and a 64 GB capacity tier, this is approximately 1.75 MB, or 0.003% of memory.

5.3 Evaluation

We evaluate Hecaton against contention-oblivious and migration-based contention-aware systems using controlled micro-benchmarks and realistic memory-intensive applications to answer four questions:

- Does mapping-based load balancing reduce convergence time and foreground interference under performance-tier contention? (§5.3.1)
- How does Hecaton behave as migration rate, working-set size, mirror capacity, and write intensity vary? Does it degrade gracefully when fewer pages can use the remapping fast path? (§5.3.2)
- Can Hecaton track short-lived and periodic contention without repeated data movement? (§5.3.3)
- Do Hecaton's benefits carry over to realistic memory-intensive applications? (§5.3.4)

Experimental setup. All experiments run on a two-socket Intel Xeon Gold 5218R platform with 40 physical cores and 80 hardware threads. Following prior work [116], we emulate a two-tier memory hierarchy with a 32 GB performance tier and a remote-NUMA capacity tier. Unless stated otherwise, the performance tier is local memory on NUMA node 1 and the capacity tier is remote memory on NUMA node 0.

Baselines. We evaluate Hecaton against three baselines:

- **HeMem** [96]: classic hotness-based tiering with no load balancing.

- **Colloid** [116]: contention-aware migration-based load balancing on top of HeMem.
- **Colloid-s** [116]: Colloid with an optimized controller that has a static migration limit, used to study the tradeoff between convergence speed and interference. We write Colloid-s-N for Colloid-s configured with an N GB/s migration limit (e.g., Colloid-s-1).

We use the HeMem-based implementation of Colloid because it provides the fastest migration; Colloid can also be built on TPP or Memtis, but their kernel migration causes Colloid to converge 2.5× and 25× more slowly, respectively [116]. We use HeMem as the representative contention-oblivious baseline; we do not separately evaluate Nomad [126] because it does not rebalance load when contention changes the relative latency of the tiers and thus would not exercise the situations that Hecaton is designed to improve. We use the same latency-measurement approach across all systems to ensure a fair comparison.

5.3.1 Convergence and Interference Under Contention

This section shows that Hecaton solves the convergence speed vs. interference tradeoff of migration-based load balancing under performance-tier contention. We compare Hecaton with HeMem, a contention-oblivious tiering system, and Colloid, a migration-based contention-aware system, under a burst of contention on the performance tier. We report workload throughput, migration traffic, and performance-tier latency; higher performance-tier latency indicates greater interference with foreground requests.

Read-only workload. Figure 5.5 shows a read-only workload whose working set fits entirely in the performance tier. Before the burst, all systems perform similarly. At t = 60 s, an antagonist increases performance-tier latency to 1.33× that of the capacity tier. The results show both why load balancing is necessary and why migration is costly. HeMem performs no load balancing, so throughput drops to 2.75 GB/s under contention. Colloid demotes

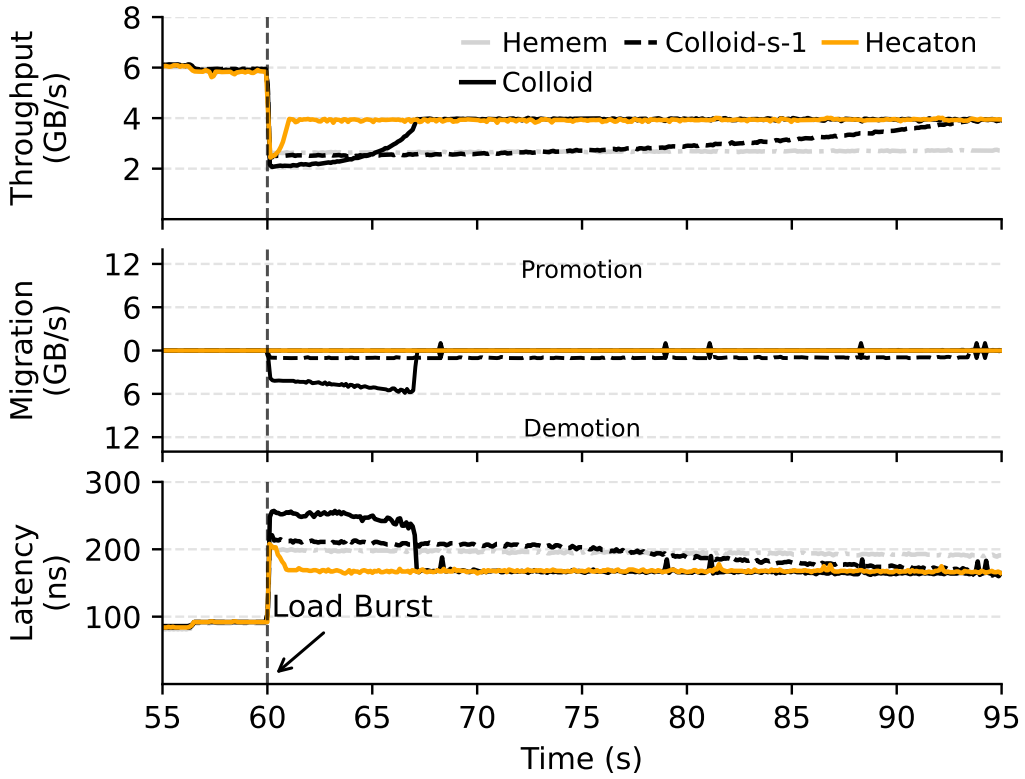


Figure 5.5: Performance tier contention: read-only workload. *A stream-like workload that uniformly randomly accesses a 32 GB read-only working set runs on the performance tier. At 60s, an 8-thread antagonist begins generating 30 GB/s of traffic (75% of performance tier bandwidth). Subplots show throughput (top), migration traffic (middle), and performance tier latency (bottom).*

pages to the capacity tier and eventually achieves 1.44× higher throughput than HeMem, but aggressive migration interferes with foreground requests: Colloid raises performance-tier latency by 1.28×. Reducing the migration rate avoids this interference but slows convergence: Colloid-s-1 keeps latency increase under 1%, but takes 33.5 s to converge. In contrast, Hecaton converges in 1.02 s, 32.8× faster than Colloid-s-1, while adding minimal foreground interference.

Read/write mixed workload. Figure 5.6 shows a 95/5 read/write workload, matching the read-heavy behavior of common applications summarized in Table 5.2. We configure Colloid-s and Hecaton with the same 1 GB/s

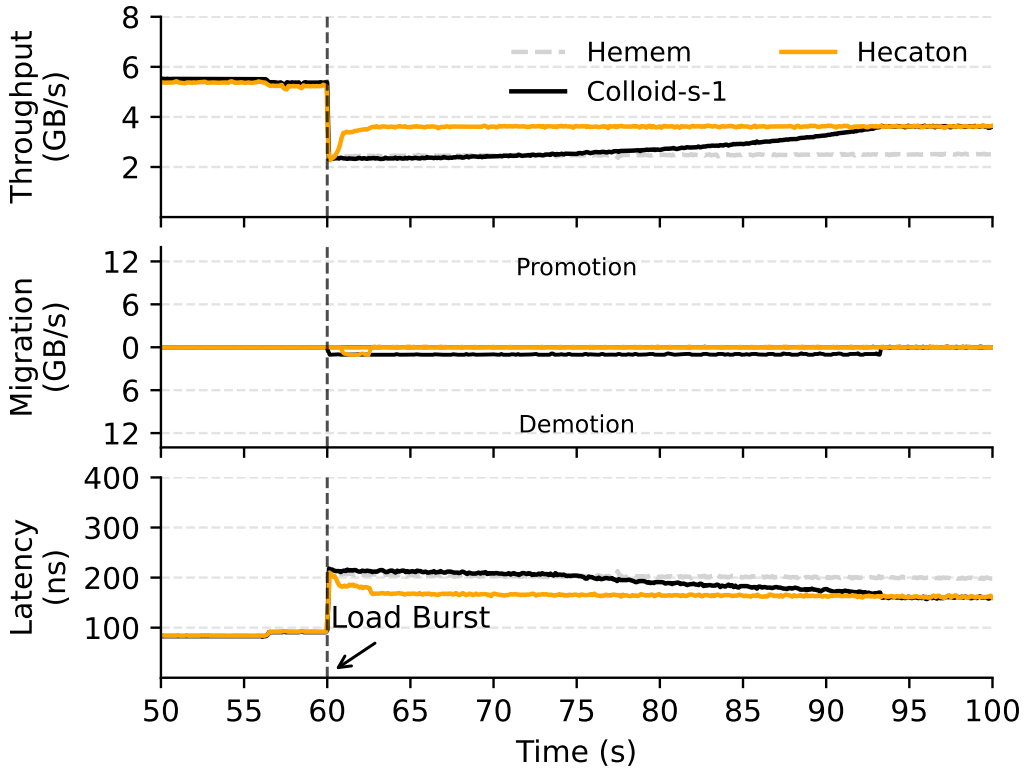


Figure 5.6: Performance tier contention: 95/5 read/write workload. *Same antagonist setup as Figure 5.5, but the workload mixes 95% reads and 5% writes over a 32 GB working set with a 5% write set. Subplots show throughput (top), migration traffic (middle), and performance tier latency (bottom).*

migration limit to isolate the benefit of remapping. The results show that Hecaton retains its advantage even when some pages cannot use the remapping fast path. After contention begins, both Colloid-s and Hecaton eventually outperform HeMem by 1.43×, but Hecaton reacts much faster: it remaps the read set first, improving throughput within 0.80 s, and migrates only the remaining 3.36 GB write set. As a result, Hecaton converges 8.2× faster than Colloid-s.

Summary. These results show that load balancing is necessary under memory-interconnect contention, but migration-based balancing faces a convergence/interference tradeoff. Hecaton avoids this tradeoff by remapping clean mirrored pages: it converges 32.8× and 8.2× faster than the

least-interfering Colloid variant on the read-only and read/write-mixed workloads, respectively, while adding minimal foreground interference.

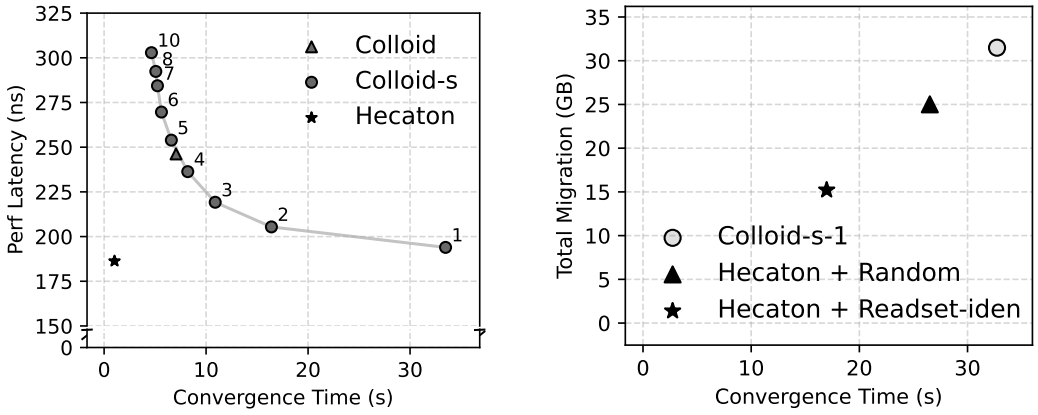
5.3.2 Sensitivity and Graceful Degradation

This section shows where Hecaton's benefits come from and how it behaves when fewer pages can use the remapping fast path. We vary migration rate, working-set size, mirror capacity, write intensity, and read-set identification while using the same setup as Figure 5.5. Figures 5.7(a) and 5.8 summarize the results: lower-left is better, indicating faster convergence with either lower performance-tier latency during convergence (Figure 5.7(a)) or less migrated data (Figure 5.8). We define convergence time as the time from a load change until the balancer reaches equilibrium, meaning that no further rebalancing is needed.

Migration limit. Figure 5.7(a) shows the convergence time vs. interference tradeoff for migration-based balancing. Increasing Colloid-s's migration limit improves convergence time but raises performance-tier latency; the most aggressive setting still converges 4.5× slower than Hecaton while raising latency by 1.58×. In contrast, Hecaton shifts most load through remapping, so it converges quickly with minimal interference. Overall, Hecaton converges 32.8× faster than the lowest-interference Colloid variant.

Read set size. Figure 5.8(a) shows how convergence time scales with read-set size. Colloid-s-1 must copy the working set, so its convergence time grows 8.1×, from 4.13 s at 4 GB to 33.5 s at 32 GB. Hecaton instead shifts load primarily through remapping, so its convergence time stays nearly flat, increasing only from 0.42 s to 1.02 s over the same range. As a result, Hecaton is 9.8× faster at 4 GB and 32.8× faster at 32 GB, with the gap widening as the read set grows.

Mirror capacity. Figure 5.8(b) shows that Hecaton degrades gracefully as mirror capacity decreases. Colloid-s-1 must migrate the entire working set regardless of available mirror space. In contrast, Hecaton remaps the mirrored portion and migrates only the unmirrored portion, so its migration



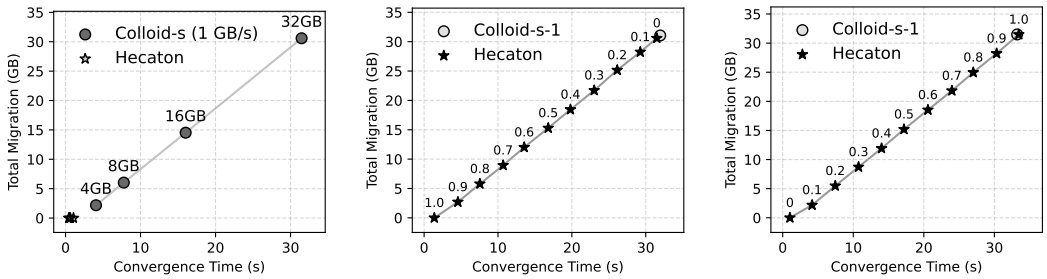
(a) Migration limit **(b) Read-set identification**

Figure 5.7: Sensitivity study: convergence vs. interference and read-set identification. *Same experiment setup as Figure 5.5. x-axis: convergence time. (a) y-axis: performance-tier latency during convergence; sweeps Colloid-s's migration limit (1–10 GB/s). (b) y-axis: migration volume; Hecaton with vs. without read-set identification at 50% write set and 50% mirroring space.*

volume grows with the fraction of the read set that has not been mirrored. When no mirror remains, Hecaton falls back to full migration and matches Colloid-s-1.

Write set percentage. Figure 5.8(c) shows that Hecaton also degrades gracefully as the write set grows. Colloid-s migrates all pages regardless of write behavior, so its convergence time remains 33.2 s. Hecaton remaps the read-only portion and migrates only the write set, so its migration volume grows with the write-set fraction. At a 10% write set, Hecaton converges in 4.2 s, 7.9× faster than Colloid-s; at a 100% write set, Hecaton migrates the full working set and matches Colloid-s.

Read-set identification. Figure 5.7(b) shows that Hecaton's write-recency test selects useful mirrors under limited mirror capacity. We use a workload with a 50% write set and restrict mirror capacity to 50% of the working set, so an ideal policy would mirror the read-only half and leave the write set for migration. We compare against a variant of Hecaton that chooses mirroring candidates randomly instead of using write-recency-based read-set identification. Hecaton approaches the ideal: it converges in 17.0 s and



(a) Read set size **(b) Mirror capacity** **(c) Write set percentage**

Figure 5.8: Sensitivity study: migration volume. *Same experiment setup as Figure 5.5. x-axis: convergence time; y-axis: migration volume during convergence. (a) varies working set (4–32 GB). (b) varies Hecaton's mirroring space (1.0×–0× WS). (c) varies write-set percentage (0–1.0).*

migrates only the write set. The random variant wastes about half of its mirror space on pages that are quickly dirtied, so those pages cannot use the remapping fast path when contention occurs; as a result, it migrates more data and converges more slowly.

Summary. These experiments show that Hecaton's advantage comes from the remapping fast path and that it degrades gracefully when that fast path is limited. Unlike migration-based balancing, Hecaton can improve convergence time without adding foreground interference. Its convergence time stays nearly flat as the read set grows, and its migration volume scales with the portion of the workload that cannot be remapped: unmirrored pages when mirror capacity is limited, and written pages as the write set grows. When no pages can use the fast path, Hecaton falls back to Colloid-s-1 behavior. Finally, the read-set identification algorithm identifies useful mirrors under space constraints, approaching the ideal and outperforming random mirror selection.

5.3.3 Tracking Bursty Contention

We now evaluate whether Hecaton can track short-lived and repeated changes in contention. These workloads stress a key weakness of migration-based balancing: pages demoted during a burst may be promoted back after the

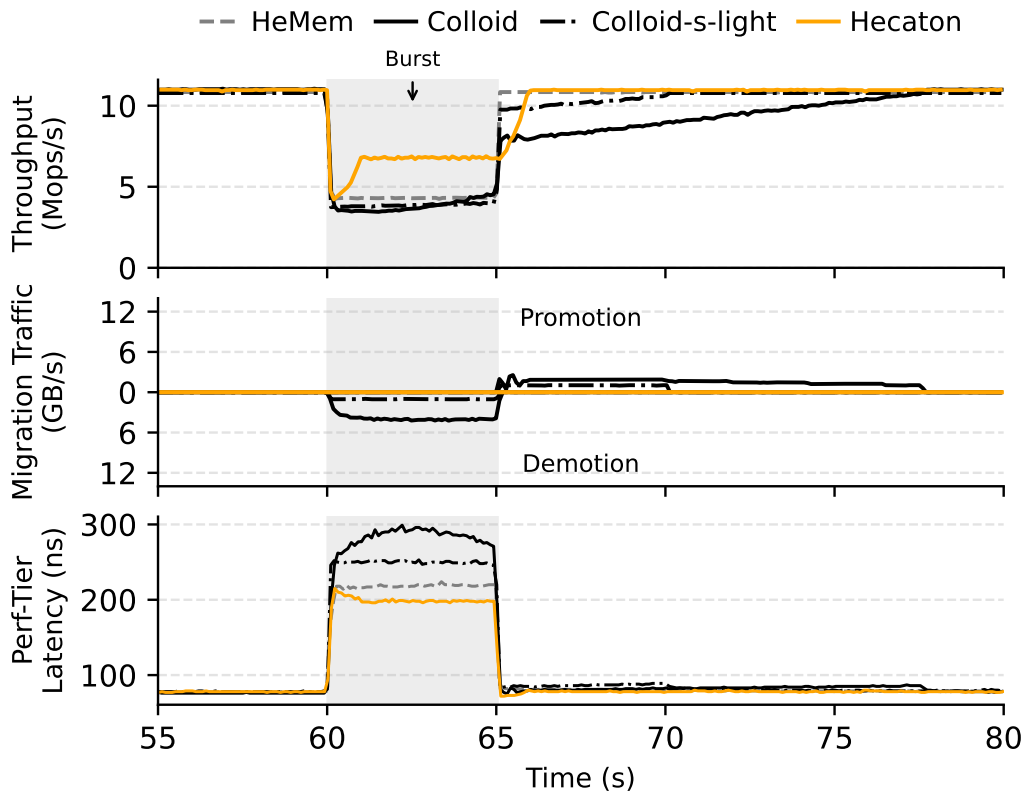


Figure 5.9: Load burst. *A multichase workload with a 32 GB working set under a single 5-second burst of contention on the performance tier. The gray region marks the burst period.*

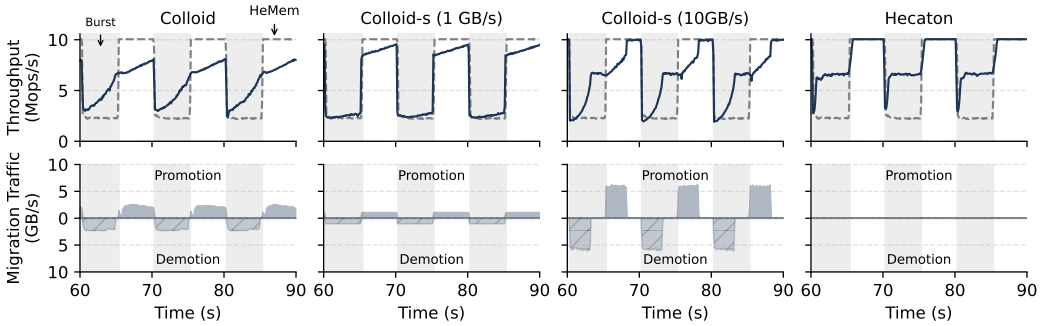
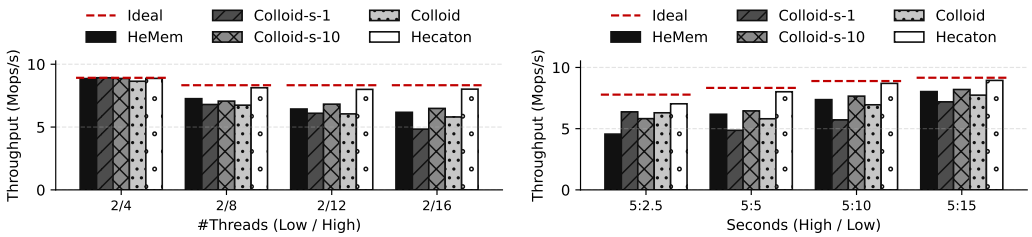


Figure 5.10: Periodic bursts. *The workload has a 16 GB resident working set. Each burst lasts 5 s and repeats every 10 s, with 16 antagonist threads active during bursts and 2 during idle periods. Gray regions mark burst periods.*

burst ends.



(a) Burst Intensity. **(b) Burst Frequency.**

Figure 5.11: Dynamic workload throughput. *Steady-state throughput of the multichase workload under periodic bursts as Figure 5.10. (a) Varies burst intensity from 2/4 to 2/16 (low-load/high-load threads) with fixed burst frequency (5s burst / 5s idle). (b) Varies burst frequency from 5s burst / 2.5s idle to 5s burst / 15s idle with fixed intensity at 2/16. Across all settings, Hecaton migrates 19–21 GB in total, while Colloid-s-1, Colloid-s-10, and Colloid migrate up to 131, 540, and 138 GB, respectively.*

Single burst. Figure 5.9 shows that Hecaton adapts to a short burst while migration-based systems do not. Both Colloid-s and Colloid converge more slowly than the burst lasts, delivering 10.3% and 11.3% lower throughput than HeMem during the burst. They also create recovery work: pages demoted during the burst must be promoted back after it ends, causing Colloid-s and Colloid to deliver 8.1% and 15.1% lower throughput than HeMem during recovery. In contrast, Hecaton adapts within 1.02 s and delivers 1.45× higher throughput than HeMem, or 1.64× higher than the best-performing Colloid variant, during the burst. Because Hecaton remaps mirrored pages rather than migrating them, it avoids the demote-then-promote pattern and adds little foreground interference.

Periodic bursts. Figure 5.10 shows that Hecaton continues to track contention when bursts repeat. Migration-based systems pay a recovery cost after each burst, because pages demoted during the burst must be promoted back afterward; as a result, Colloid-s and Colloid achieve only 0.73–0.84× HeMem's throughput during idle periods. Hecaton instead remaps mirrored pages across phases, achieving 2.60× HeMem's throughput during bursts and 0.98× HeMem during idle periods. It also migrates 25.7× less data than the best-performing Colloid variant (21 GB vs. 540 GB for Colloid-s-10), because

remapped pages can be remapped back rather than physically moved again.

Dynamic-workload throughput. Figure 5.11 shows how Hecaton performs across a range of burst intensities and frequencies. We compare against an idealized upper bound with instantaneous, cost-free placement: pages are placed on the capacity tier during bursts, when the performance tier is slower, and on the performance tier during idle periods. We compute ideal throughput by measuring each placement separately and combining the results in proportion to the burst and idle durations.

Figure 5.11(a) varies burst intensity. Hecaton stays close to the ideal, while both HeMem and Colloid degrade as bursts intensify. HeMem performs no load balancing, so it falls to 0.74× the ideal at the highest intensity. Colloid and its variants do balance, but slow convergence is penalized more under stronger contention; even the best Colloid variant reaches only 0.76× the ideal. Hecaton, in contrast, reaches 0.96× the ideal at the highest intensity because mapping-based load balancing converges fast enough to keep up with load changes.

Figure 5.11(b) varies burst frequency. Hecaton stays close to the ideal across all settings, reaching 0.90× the ideal even when bursts arrive every 2.5 s and 0.98× the ideal at the lowest frequency. HeMem improves as bursts become less frequent, from 0.59× to 0.88× the ideal, because more time is spent in the unloaded phase. Colloid variants remain limited by slow convergence, interference, and recovery traffic: even at the lowest frequency, the best Colloid-s variant reaches only 0.79× the ideal, while Colloid drops to 0.71× as longer idle periods trigger extra promotion before the next burst.

Summary. Across all burst-intensity and burst-frequency settings, Hecaton stays within 0.90–0.99× the ideal, while HeMem and the Colloid variants drop as low as 0.58×. Hecaton outperforms the best baseline by 1.10–1.29× in seven of eight settings and matches it in the low-contention case, where all systems run near the ideal. Hecaton also migrates 6–26× less data than any Colloid variant.

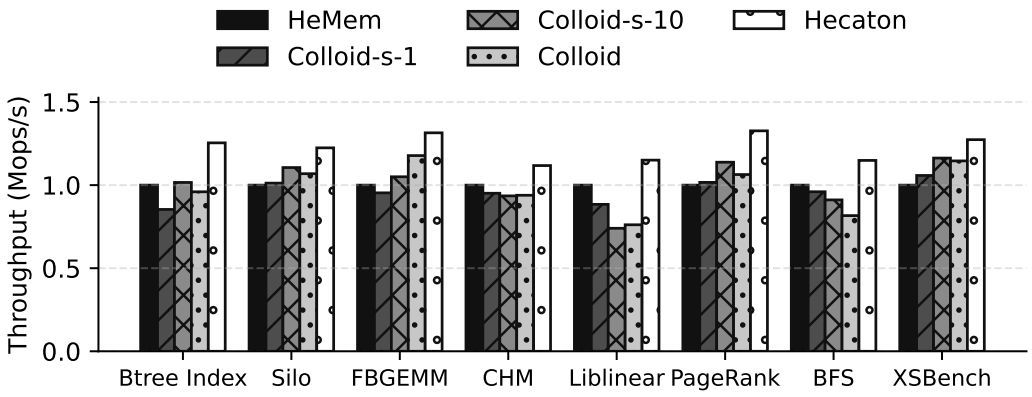
Category	Workload	WS (GB)	Read %	Skew	MLP	Access pattern
DB	Btree Index [64]	16.7	99.9	High	Low	Pointer chase
	Silo [115]	14.1	99.6	High	Low	Pointer chase
ML	FBGEMM [81]	17.2	99.8	Med	High	JIT + prefetch
	CHM [110]	19.6	99.6	Low	Low	Hash chain
	Liblinear [39]	32.6	96.9	Low	Med	Sequential scan
Graph	PageRank [21]	16.7	95.3	High	High	Vectorized
	BFS [21]	19.0	96.1	Low	High	Vectorized
HPC	XSBench [113]	16.7	99.9	High	Med	Random lookup

Table 5.2: Realistic workload characteristics. *All workloads use 4 application threads except btree, which is single-threaded. Stack and other small, bounded-size kernel data structures are not managed by HeMem and are allocated in DRAM; their footprint is negligible.*

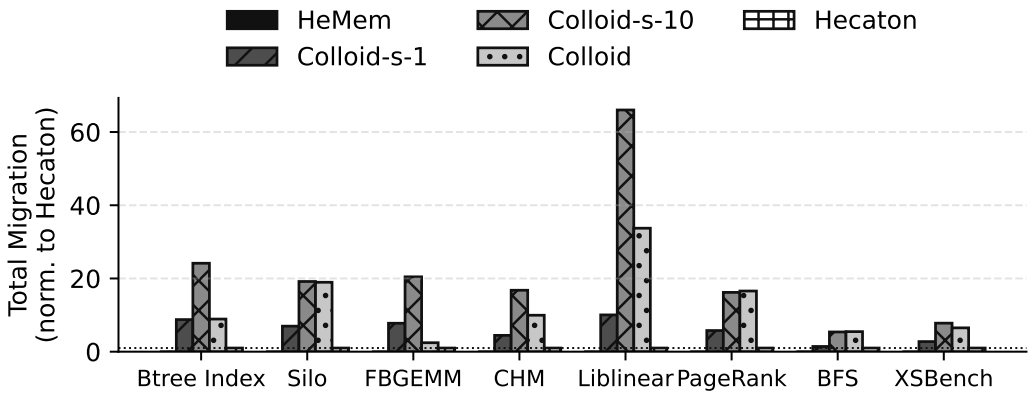
5.3.4 Realistic Memory-Intensive Workloads

Hecaton's benefits apply to realistic memory-intensive workloads. We evaluate eight representative workloads widely used in tiered-memory evaluation [64, 96], including ML workloads from the DCPerf datacenter benchmark suite [110]. As shown in Table 5.2, these workloads span database indexes, ML kernels, graph analytics, and HPC, with different working-set sizes, read fractions, access skews, memory-level parallelism, and access patterns. Each workload runs against the same periodic-burst antagonist used in Section 5.3.3. Figure 5.12(a) reports throughput normalized to HeMem, and Figure 5.12(b) reports total migrated data normalized to Hecaton.

Hecaton improves throughput across all eight workloads, outperforming HeMem by up to 1.33× and the best Colloid variant by up to 1.30×. Across categories, Hecaton improves over HeMem by 1.22–1.25× on database indexes, 1.12–1.31× on ML workloads, 1.15–1.33× on graph workloads, and 1.27× on XSBench. Against the best Colloid variant, the corresponding improvements are up to 1.23×, 1.30×, 1.20×, and 1.10×. Hecaton also reduces data movement substantially, migrating 26 GB on average compared to 358 GB for Colloid, and up to 34× less in the Liblinear experiment.



(a) Throughput normalized to HeMem.



(b) Total data migrated (GB).

Figure 5.12: Realistic workloads under periodic bursts. *Same burst setup as Figure 5.10. (a) Steady-state throughput normalized to HeMem. (b) Total data migrated during the experiments normalized to Hecaton.*

Hecaton benefits most when workloads have large read-mostly working sets that are costly for migration-based systems to move quickly. Liblinear illustrates this case: it has the largest working set in our suite (32.6 GB), and a large portion of its dataset is scanned uniformly during each training iteration. As a result, migration-based systems struggle to rebalance it quickly, while Hecaton can remap the read-mostly portion and migrate only pages that cannot use the fast path. BFS shows a similar effect: it runs on a uniform random graph whose 19.0 GB working set is essentially all hot, so Hecaton's

advantage grows with hot-set size, as in Figure 5.8(a).

The gains are smaller when migration-based systems already have an easier balancing problem or when the workload tolerates memory latency. For highly skewed workloads, such as PageRank and XSBench, much of the access load is concentrated on a smaller hot set, so Colloid can adapt with less migration. XSBench also has high memory-level parallelism, which helps mask higher memory latency under contention. FBGEMM similarly exposes high memory-level parallelism through JIT-compiled kernels and prefetching; even when Colloid leaves much of the working set on the capacity tier, this parallelism hides some of the slowdown and narrows Hecaton's relative gain.
Summary. Across realistic workloads, Hecaton's advantage follows the same principles as the controlled experiments: remapping helps most when the read-mostly working set is large and difficult to migrate quickly, and helps less when access skew or memory-level parallelism reduces sensitivity to placement. Even in the less favorable cases, Hecaton matches or outperforms the best migration-based baseline while migrating far less data.

5.4 Conclusion

We introduced Hecaton, a contention-aware heterogeneous memory management system designed for contended memory. Hecaton introduces mapping-based load balancing, a low-cost alternative to migration that shifts load by updating page table mappings rather than copying data. Under memory-interconnect contention, Hecaton outperforms contention-oblivious tiering and, compared to migration-based contention-aware systems, adapts to bursty load with faster convergence, reduces migration writes, and adds minimal interference to foreground workloads. Across microbenchmarks, dynamic workloads, and eight realistic applications, Hecaton converges up to 32.8× faster than the lowest-interference Colloid variant, migrates up to 34× less data, and improves throughput over the best baseline by up to 1.30×. As memory bandwidth becomes an increasingly critical resource [54], we believe contention-aware load balancing will become an essential property

of heterogeneous memory systems.

Chapter 6

Related Work

In this chapter, we discuss how prior work relates to the three parts of this dissertation. First, we discuss work related to our Non-Hierarchical Caching project (Section 6.1): algorithms and policies for hybrid storage systems, storage optimization, storage-aware caching and tiering, and the management of NVM-based devices. Next, we discuss work related to MOST (Section 6.2): tiered memory management, tiered storage systems, distributed multi-tier caching, machine-learning-based tiering, and storage-aware caching and tiering. Finally, we discuss work related to Hecaton (Section 6.3): software- and hardware-based heterogeneous memory management, NUMA management, and the storage-tiering analogs from which Hecaton draws inspiration.

6.1 Optimizing Caching for Modern Non-Hierarchical Storage

This section discusses prior work related to Chapter 3, where Orthus introduces Non-Hierarchical Caching (NHC) for storage hierarchies whose neighboring tiers have overlapping performance characteristics. We organize the discussion around algorithms and policies for hybrid storage systems, storage optimization, storage-aware caching and tiering, and the management

of NVM-based devices.

Algorithms and Policies in Hybrid Storage Systems. Algorithms and policies for managing traditional storage hierarchies have been studied extensively [62, 63, 118, 120, 121, 135, 137]. Techniques have been introduced to optimize data allocation [62, 118, 121, 135], identify hot data, and perform data migration [63, 120, 137]. Most previous work improves performance by focusing on workload access locality. In contrast, NHC improves performance by taking all devices and the current workload into account, redirecting load to the capacity tier when the performance tier saturates.

Storage Optimization. A long line of pioneering work in storage management [40, 62, 121] 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 while leaving longer-term optimization to a higher-level system.

Storage-Aware Caching/Tiering. Orthus shares aspects with storage-aware caching and tiering [20, 32, 59, 88, 93, 124, 125], which considers more factors than hit rate. For instance, Oh et al. [88] propose over-provisioning in Flash to avoid the influence of SSD garbage collection. Modern devices such as NVDIMM and Optane SSD have distinctive characteristics compared to Flash, and we study the implications of these emerging devices for caching and tiering. BATMAN [32] shares Orthus's motivation: classic caching is ineffective when the bandwidth of the capacity layer is a significant fraction of overall bandwidth. However, it investigates a much simpler hierarchy with a fixed performance difference (4:1 between high-bandwidth memory and DRAM); given this fixed difference, BATMAN splits cache accesses between HBM and DRAM statically. This does not generalize to modern hierarchies where performance differences vary dynamically with the amount of writes or the level of parallelism in the workload. Wu et al. [125] study tiering on SSDs and HDDs and recognize a similar problem: SSDs, or faster devices, can become the throughput bottleneck. To mitigate it, they periodically migrate data from SSDs to HDDs when the SSD response time exceeds that of the

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, like SIB [59], it estimates workload intensity over a period and migrates data based on the estimate, so it struggles with dynamic workloads. Third, it is tuned for a specific hierarchy (SSDs and hard drives). Unlike these approaches, NHC focuses on improving caching, adapts its behavior at runtime, reacts to complex and dynamic workloads, and works well across a range of modern devices.

Managing NVM-based Devices. Another line of work integrates NVM-based devices into the memory–storage hierarchy and proposes file systems and databases tailored to NVDIMM and low-latency SSDs. Strata [63] is a cross-media file system that mitigates the limitations of NVM, SSD, and HDD in a traditional storage hierarchy, and Ziggurat [137] tiers data across DRAM, NVM, and SSD. Spitfire [139] is a three-tier caching-based buffer manager that leverages machine learning for data placement. Dulloor et al. [37] and others [62, 118, 135] propose NVM-aware data placement strategies that optimize placement over longer periods (offline or periodically); NHC complements them by handling more dynamic workload changes at runtime. Orthus is the first to analyze general caching and tiering on modern hierarchies through modeling and empirical evaluation, and the first to propose a generic solution, NHC, to realize the full performance benefits of such a hierarchy.

6.2 Mirror-Optimized Storage Tiering

This section discusses prior work related to Chapter 4, where MOST combines tiering with selective mirroring to enable cross-tier load balancing in heterogeneous storage hierarchies. We organize the discussion around tiered memory management, tiered storage systems, distributed multi-tier caching, machine-learning-based tiering, and storage-aware caching and tiering.

Tiered Memory Management. MOST shares scope with research on tiered memory and storage [16, 37, 48, 62, 64, 77, 94, 96, 108, 118, 121, 135] for managing heterogeneous hierarchies of different devices, including DRAM, NVM, CXL-enabled memory, NVMe SSD, SATA SSD, and hard drives. vTMM [105], a tiered memory management system for virtual machines, optimizes memory-access tracking through page-modification logging. Nimble [128] is an OS-level system designed to improve page migration throughput. HeterOS [57] functions as an OS/VMM-level heterogeneous memory manager, coordinating memory placement and migration with guest OSes. Thermostat [18] identifies hot memory pages using page table sampling. All of these systems are orthogonal to MOST, as they focus on memory-access tracking, hot/cold data identification, and data migration rather than load balancing across tiers.

Tiered Storage Systems. Strata [63] is a cross-media file system designed to mitigate the limitations of NVM, SSD, and HDD in a traditional storage hierarchy. Ziggurat [137] migrates cold data to lower tiers in an NVM–disk tiered file system. Spitfire [139] is a three-tier caching-based buffer manager that leverages machine learning for optimal data placement. AutoRAID [120] integrates mirroring for improved write performance with RAID-5 for cost-effective capacity. hStorage-DB [73] enforces QoS policies for requests in hybrid storage systems. However, these systems are load-unaware and not able to fully utilize lower-tier bandwidth. PolyStore [100] is a meta layer built on top of storage-medium-optimized file systems that maximizes cumulative storage bandwidth. However, PolyStore is optimized exclusively for file systems and does not maintain cross-tier duplication, therefore not able to efficiently adapt to dynamic workloads.

Distributed Multi-tier Caching. Distributed multi-tier caching systems [72, 95, 99] broadly share a similar problem space with MOST. However, their approaches differ significantly from MOST in both design and objectives. For instance, EC-CACHE [95] uses erasure coding instead of mirroring and is limited to load balancing within a single tier. ELECT [99] replicates

data within a single tier and lacks support for cross-tier load balancing. eMRC [72] focuses on multi-tier miss ratio approximation but does not address the load balancing problem, making its objective orthogonal to ours. In contrast, MOST introduces a novel data management model that supports selective mirroring and adaptive load balancing across a heterogeneous storage hierarchy.

Machine-Learning-based Tiering. Machine learning-based storage and memory tiering approaches (e.g., IDT [28], RLTiering [70], ArtMem [132], and autonomous data tiering for hierarchical storage [90]) target a similar problem space as MOST. These systems typically utilize reinforcement learning to guide data placement decisions but are built on traditional single-copy tiering models. As shown in our evaluation of Colloid in Chapter 4, such models struggle with dynamic and time-varying workloads. Like Colloid, these ML-based approaches are fundamentally constrained by the inherent limitations of single-copy designs and fail to address the critical challenge of load balancing across tiers in heterogeneous hierarchies.

Storage-Aware Caching/Tiering. MOST aligns with research on storage-aware caching and tiering [20, 59, 93, 123–125]. Wu et al. [125] identified a similar issue, where SSDs become throughput bottlenecks, and proposed migrating data to HDDs when SSD response times exceed those of HDDs. However, this migration-only approach reacts slowly to workload changes and incurs high migration overhead, like Colloid. SIB [59] targets HDFS clusters with SSDs and HDDs, using SSDs as write buffers and offloading reads to HDDs. LBICA [20] implements cache load balancing by halting new allocations to performance devices under burst loads but does not balance read traffic. Like Orthus, these caching-based systems fail to fully utilize the capability of modern storage hierarchies.

6.3 Heterogeneous Memory System

This section discusses prior work related to Chapter 5, where Hecaton introduces a mapping-based mechanism for load balancing in heterogeneous memory hierarchies. We organize the discussion around software-based heterogeneous memory management, hardware-based heterogeneous memory management, NUMA management, and the storage-tiering analogs from which Hecaton draws inspiration.

Software-based Heterogeneous Memory Management. Hecaton shares scope with research on software-based heterogeneous memory management [18, 29, 32, 56–58, 64, 68, 69, 77, 96, 98, 107, 116, 126–128]. For example, SoarAlto [68] places pages by performance criticality rather than hotness, based on the observation that memory-level parallelism can mask latency, so frequently accessed pages are not necessarily performance-critical. PACT [47] extends the idea of performance criticality to per-page granularity. Memtis [64] maintains a histogram of page access counts and uses access-distribution-aware estimation to dynamically classify hot and cold pages, while also adapting the page size between 4 KB and 2 MB to balance tracking accuracy and overhead. These systems innovate on mechanisms for access tracking (e.g., HeMem [96]), page migration (e.g., Nimble [128]), dynamic page-size determination (e.g., Memtis [64]), and thrashing mitigation (e.g., Nomad [126]), but do not optimize load balancing under contention.

Hardware-based Heterogeneous Memory Management. Hardware-managed tiering moves page-placement decisions into the hardware by treating the performance tier as a cache for the capacity tier, enabling data placement at cache-line granularity. Intel's Memory Mode uses an inclusive cache, while Intel's Flat Memory Mode uses an exclusive cache. Zhong et al. [138] build on this with a software runtime that identifies tiering-sensitive VMs and protects them from interference in multi-tenant environments. PIPM [49] co-designs hardware page migration with the multi-host CXL coherence protocol to enable fine-grained, incremental migration in disaggregated shared memory. However, these hardware-managed systems make the same assumption as

their software counterparts: the performance tier is always faster than the capacity tier. As a result, they cannot react to latency inversion. It would be interesting to integrate mapping-based load balancing with hardware-based tiered memory management to balance load through page table remapping at cache-line granularity.

NUMA Management. Hecaton shares similar scope with NUMA management [17, 24, 26, 31, 36, 74]. In NUMA systems, pages are duplicated across NUMA nodes to improve locality but not for load balancing [26]. Carrefour [36] generalized NUMA placement as traffic management, selecting among migration, replication, and interleaving based on locality and congestion. StarNUMA [31] introduced a shared memory pool for heavily shared pages that have no good home on any single socket. MEMPLEX [74] showed that replication and migration are complementary in multi-chiplet systems where sharing makes single-copy placement fundamentally insufficient. These systems replicate pages for locality, while Hecaton uses redundancy for a new purpose: load balancing.

Heterogeneous Storage Management. Hecaton shares scope with storage tiering systems [100, 114, 137]. For example, PolyStore [100] exploited the combined bandwidth of multiple devices in a storage hierarchy with transparent, fine-grained data placement. MOST [114] mirrors a small hot set across tiers to balance load through routing while keeping the rest single-copy. However, as we showed in §5.1, this routing-based balancing is not applicable to memory systems where hardware directly resolves virtual-to-physical address mappings. These storage-based systems are either single-copy, like PolyStore, requiring migration to balance load, or routing-based, like MOST, which cannot be applied to memory systems.

Chapter 7

Conclusion and Future Work

In this closing chapter, we summarize each part of the dissertation (§7.1), discuss potential future work directions (§7.2), comment on general experiences and lessons learned from this research journey (§7.3), and finally conclude (§7.4).

7.1 Summary

This dissertation identifies the non-hierarchical characteristics of modern heterogeneous storage and memory systems, where the performance gap between neighboring devices has narrowed to within a small factor. Based on this observation, the dissertation comprises three parts that revisit classic and state-of-the-art approaches for modern heterogeneous storage and memory systems. In the first part of the dissertation, we revisit classic caching on modern heterogeneous storage systems, show that classic caching leaves the capacity device under-utilized, and propose Non-Hierarchical Caching (NHC), which dynamically redirects a fraction of read hits to the capacity device under saturation. NHC outperforms prior state-of-the-art approaches across a range of static and dynamic workloads. In the second part, we step back and ask how a collection of devices with similar performance should be managed collectively to deliver both high performance and capacity efficiency. Revisiting classic approaches such as tiering, striping, and mirroring, we

find that mirroring a small amount of hot data improves load balancing. We therefore introduce Mirror-Optimized Storage Tiering (MOST), which combines the capacity efficiency of tiering with the load-balancing flexibility of mirroring: MOST mirrors a small amount of hot data across tiers so the system can balance load by routing rather than by migration. In the third part, we extend these ideas to heterogeneous memory systems and introduce mapping-based load balancing, a faster, low-cost alternative to migration. We implement mapping-based load balancing in Hecaton, a tiered-memory system that mirrors read-intensive pages across tiers and balances load by remapping page tables rather than migrating data, achieving much faster convergence and lower interference when the memory system is contended. The following sections summarize each part in detail.

7.1.1 Optimizing Caching for Modern Non-Hierarchical Storage

In the first part of this dissertation, we address the capacity under-utilization problem of classic caching by introducing Non-Hierarchical Caching (NHC), which augments any classic cache controller with a lightweight, feedback-driven scheduler that dynamically adjusts the fraction of read hits redirected to the capacity device. When the performance device is lightly loaded, NHC behaves like classic caching, maximizing the hit ratio; when it saturates, NHC offloads excess traffic to exploit the capacity device's bandwidth, requiring no prior workload or device knowledge. We implement NHC as Orthus-CAS, a kernel module within Open CAS [6], an open-source block-layer caching framework. We compare NHC with two state-of-the-art approaches, SIB and LBICA, by faithfully re-implementing both within Open CAS. NHC outperforms LBICA on both static and dynamic workloads because LBICA does not redirect any cache hits. NHC also outperforms SIB and its two variants on dynamic workloads with varying write ratios, demonstrating its ability to adapt to non-stationary workloads.

7.1.2 Mirror-Optimized Storage Tiering

In the second part of this dissertation, we revisit several classic approaches for managing non-hierarchical storage systems—striping, caching, tiering, and mirroring. We classify these approaches by the number of data copies they maintain into two families: single-copy and multi-copy. We find that classic single-copy approaches (e.g., striping and tiering) can adjust load distribution only by migrating data across tiers. However, migration is slow, generates extra writes that shorten device lifetime, and interferes with foreground workloads. Multi-copy approaches such as caching and mirroring, on the other hand, can adjust load distribution by rerouting requests to different copies without data migration. However, these methods are capacity-inefficient because the duplicated copies waste performance-tier capacity.

To combine the benefit of both, we introduce Mirror-Optimized Storage Tiering (MOST), which adds a small amount of mirroring on top of classic tiering. Mirrored copies of hot data give the host the freedom to route requests dynamically: under low load, MOST behaves like a classic tiering system that maximizes performance-tier utilization; under high load, it routes a tunable fraction of requests to the capacity-tier copy to harvest idle bandwidth. Because routing is cheap and reversible, MOST reacts quickly to workload changes, reduces device writes by avoiding unnecessary migrations, and is robust to performance fluctuations within a single device. We implement MOST as Cerberus, a storage management layer within CacheLib. Across static cache workloads on two storage hierarchies, Cerberus achieves up to 2.34× higher throughput and up to 75% lower P99 latency than state-of-the-art systems including Colloid, HeMem, and Orthus; across four production cache workloads, it improves throughput by up to 1.86×, reduces P99 GET latency by up to 90%, and reduces device writes by up to 84% under dynamic workloads.

7.1.3 Mapping-Based Load Balancing

In the third part of this dissertation, we address load balancing in heterogeneous memory systems with a small, fast performance tier and a larger capacity tier, where memory-interconnect contention can make the performance tier slower than the capacity tier. Existing contention-aware systems respond to this latency inversion by migrating pages across tiers, but migration is the wrong mechanism under contention: it adds traffic to the already-contended memory subsystem, slows convergence, interferes with foreground requests, and repeats data movement under bursty load changes.

To address this, we introduce mapping-based load balancing and build Hecaton, a tiered-memory system that implements it on top of Colloid. Hecaton maintains mirrored copies of read-mostly pages across tiers and shifts load by changing page-table mappings rather than copying data. To make this practical, Hecaton uses a lightweight latency-feedback controller, low-overhead reactive and proactive mirroring, and a write-recency test that identifies pages whose mirrors are likely to remain clean. Hecaton reacts quickly to dynamic contention while avoiding the data movement and interference of migration-based systems. Across microbenchmarks, bursty workloads, and realistic memory-intensive applications, Hecaton delivers up to 2.60× the throughput of contention-oblivious tiering under performance-tier contention, converges up to 32.8× faster than the lowest-interference Colloid configuration, achieves up to 1.64× higher throughput than the best-performing migration-based baseline, and migrates up to 34× less data.

7.2 Future Work

7.2.1 Sustainability in Heterogeneous Storage System with High-Capacity SSD

With today's development of large language models, the amount of context memory is increasing dramatically. This context memory stresses existing

storage systems and is increasingly offloaded to SSDs as a cheaper yet fast alternative to DRAM. High-capacity SSDs are an appealing fit for this scenario. High-capacity SSDs are a class of drives that pack large capacity onto a single device by storing more bits per flash cell. However, high-capacity SSDs raise a sustainability concern, another critical challenge in today's datacenters. Datacenters must reduce embodied carbon emissions, particularly from flash, which accounts for 40% of embodied carbon in servers [78]. As we discuss below, the limited endurance and large indirection units of high-capacity SSDs shorten device lifetime, making their carbon footprint particularly difficult to reduce.

The limited endurance of high-capacity SSDs stems from their underlying high-density flash cells, which encode multiple bits per cell. A cell encoding n bits represents 2^n distinct voltage states—one bit yields two states, four bits yield sixteen. Today's densest commercial flash uses QLC, which stores four bits per cell, and the latest commercial product [9] offers up to 245 TB of capacity in a single drive. Higher density, however, comes at a cost in endurance. Each additional bit per cell reduces the number of times a flash cell can be rewritten before failing by roughly an order of magnitude. For example, the largest-capacity drive offers only 0.3 drive writes per day (DWPD), meaning it can sustain writing 0.3 of its full capacity per day over a 5-year warranty. By comparison, a typical SLC SSD sustains 60 DWPD, roughly 200× higher. Moreover, high-capacity SSDs use a larger indirection unit, 16 KB on the largest-capacity drive compared to 4 KB on conventional SSDs. For example, a 1 KB write incurs 4× higher write amplification on these drives than on a conventional drive with 4 KB indirection units.

As a result, deploying these drives sustainably is challenging given their limited endurance and large indirection units. One promising approach is to pair them with high-endurance devices, forming a heterogeneous storage system that combines high capacity with high endurance. However, efficiently managing such systems with diverse endurance properties to maximize sustainability remains an open question.

7.2.2 Performance Isolation in Heterogeneous Storage and Memory Systems

In today's datacenters, servers are commonly shared by multiple tenants for better resource utilization. Performance isolation is therefore critical: tenants must share resources according to a sharing policy. One such policy is quality-of-service (QoS) [124], under which each tenant is given a latency target (typically P99) and the system manages resources to ensure no tenant's target is violated. Another is fairness, where each tenant receives a fair share of resources so no single tenant can monopolize the system.

In the three parts of this dissertation, we optimize for the performance of heterogeneous storage systems but do not address performance isolation. For example, MOST (the second part of this dissertation) mirrors a small amount of data across the two tiers without distinguishing between tenants. In the worst case, all of one tenant's data may be mirrored while other tenants' data is not, so when the performance device saturates, the majority of that tenant's load is offloaded to the capacity tier. This can violate sharing goals such as QoS, because the capacity tier typically has worse tail latency than the performance tier. It also remains unclear how to route requests across tiers in a multi-tenant setting without violating any tenant's target. The current design of MOST lacks the mechanisms to address this: it manages data transparently to applications and cannot distinguish data or requests by tenant. Designing such tenant-aware mechanisms remains an open question. A similar issue arises in the third part of this dissertation: when load is balanced by routing part of the memory accesses to the capacity tier, applications whose accesses are offloaded can experience worse tail latency, since the capacity tier's tail latency is typically higher [67].

Together, these examples reveal one key insight: the non-hierarchical characteristics of modern heterogeneous storage and memory systems make performance isolation harder. In a hierarchical system, only the performance device must be shared properly; in a non-hierarchical system, the capacity device must also be shared once it is utilized, and the load-balancing

mechanism interacts directly with the performance isolation policy. How to properly share a non-hierarchical heterogeneous system is, to our knowledge, an open question.

7.2.3 Optimizing the Software Stack for Hybrid Devices

A hybrid device is a device with internal heterogeneity, combining multiple media within a single device. One example is the CXL-SSD [133], which pairs a small amount of DRAM with a large amount of NAND flash behind a CXL controller to provide memory-like semantics. Such devices differ from existing ones because of their internal heterogeneity. Although they are not yet commercially available, a recent prototype [133] reveals one key property: a bimodal latency pattern, with low latency when requests hit the fast medium (~800 ns) and high latency when they miss to the slow medium (~10 μs). Deploying such hybrid devices in practice presents three challenges. We use in-memory caching (e.g., Redis and Memcached) as a running example, since it is a natural use case for these devices. First, device performance improves as the on-device DRAM hit ratio rises, but current in-memory caching designs are unaware of the underlying bimodal latency. How to manage data layout to maximize the on-device hit ratio remains an open question. Second, tail latency must be controlled: when a request misses in the on-device DRAM, it incurs the much higher flash latency, inflating overall tail latency. How to mitigate this worst case remains an open question. Third, the endurance of the flash cells inside the device must also be carefully managed to maintain sustainability. This requires improving the on-device hit ratio and reducing write amplification. As this example shows, adopting such hybrid devices remains challenging and requires rethinking how the software stack is built.

7.3 Lessons Learned

In this section, we describe three lessons learned across the three parts of this dissertation. Specifically, these lessons help us identify the right research problem, reason about a complex system, and build a new system systematically.

7.3.1 Measure First, then Build

Always start with measurement: measuring a system is the key to understanding its behavior, and the first step toward finding a research problem is to understand it deeply and demonstrate it clearly. Understand when and why a system behaves the way it does under a given workload. Understand why it breaks down on a new setup, such as a new device. Understand why it performs poorly on a new workload. These measurements lay the foundation for building something new that addresses the issues they reveal. This is the most important lesson we learned across the three projects in this dissertation: measure the real system, and show when and why it falls short. For example, in the third project, we started by running the state-of-the-art system under a dynamic, bursty workload not examined in the original paper. We found that it performed worse than a system that performs no load balancing at all, which exposed the inefficiency of migration-based balancing in non-hierarchical memory systems. This finding motivated our third project, which utilizes unused capacity to enable mapping-based load balancing that is faster and introduces less interference than migration.

7.3.2 Reason with Mechanisms, then Policies

While measurement reveals the problems a system has under a given scenario, it does not reveal the reasons behind them. The key to reasoning about the behavior of a complex system is separating mechanisms from policies, one of the most important principles we have learned from systems research [65]. A mechanism defines what a system is capable of doing, while a policy defines

how the system behaves based on those capabilities. From a system-design perspective, this decoupling makes a system flexible: one can change the high-level behavior (policy) without rewriting the underlying implementation (mechanism). From a research perspective, the mechanism is the right lens for understanding systems: when building a new system or comparing existing ones, identifying each system's core mechanism reveals what makes it novel, how it differs from prior systems, and how it shapes performance. We learned this lesson while working on the second part of this dissertation. In our early submissions, we struggled to distinguish our approach from prior systems and to articulate its key innovation, until we realized that the mirrored copy was the core mechanism setting it apart. Mirroring allows the system to change its load distribution freely and at low cost, so its advantage is greatest under dynamic workloads. Based on this understanding, we focused the evaluation on demonstrating these benefits. The lesson is to reason about the mechanism first, then the policy.

7.3.3 Start with Strict Assumptions, then Relax

The last lesson is about how to build a new system: start with strict assumptions, and grow the system by relaxing them one at a time. A complex system is difficult to design in one shot. Strict assumptions shrink the problem to a core that admits a simple, working design; each relaxation then adds one capability and exposes a new set of challenges, sometimes extending the existing design naturally and sometimes forcing part of it to be redesigned. Using the second part of this dissertation as an example, we began by assuming a read-only workload, under which mirroring and request routing alone are sufficient to balance load. We then relaxed this assumption to allow writes, which introduced a real challenge: a write invalidates the mirrored copy, so reads to that data can no longer be routed across mirrors, and in workloads that mix reads and writes, writes interfere with reads to the same data. Building on the existing read mechanism, we aligned the management granularity with the write granularity so that writes can be balanced just

like reads, and we added a cleaning mechanism to handle dirty copies. Each relaxation extended a working system to a more general workload. The lesson is to grow a system incrementally: choose assumptions strict enough that the initial design is simple, then relax them one at a time, extending the mechanism at each step.

7.4 Closing Remarks

Heterogeneous storage and memory systems are increasingly deployed at scale. Hyperscalers deploy CXL memory in production to address the memory wall [77, 138], and Nvidia advocates GPU-centric storage that treats high-performance SSDs as a high-bandwidth memory tier alongside HBM and DRAM, addressing the data starvation problem in modern AI [87]. As this dissertation shows, software stacks built around legacy devices must be reexamined to fully embrace these emerging devices.

This dissertation takes a step forward by identifying the non-hierarchical characteristics of modern heterogeneous systems and revisiting a range of classic approaches to managing them. We revisit classic caching and propose Non-Hierarchical Caching (NHC), which redirects a fraction of read hits to the capacity device when the performance device saturates (Chapter 3). We then revisit tiering and propose MOST, which mirrors a small set of hot blocks across tiers and routes requests dynamically to balance load without migration (Chapter 4). Finally, we study load balancing in heterogeneous memory and propose Hecaton, which remaps mirrored pages across tiers instead of moving data, reacting quickly to memory-interconnect contention (Chapter 5).

We believe the observations and approaches developed here extend beyond the specific devices and setups studied in this dissertation, and they will continue to apply as new device classes emerge. Many questions remain open: sustainability under flash endurance constraints, performance isolation across tenants, the software stack for hybrid devices with internal heterogeneity, and heterogeneity in GPU-centric storage systems. We believe

that the studies and ideas presented in this dissertation will aid in future research to address some of these challenges.

Bibliography

[1] Amazon EBS. https://aws.amazon.com/ebs/pricing/.

[2] CacheLib. https://cachelib.org/.

[3] CacheLib Cachebench. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_Overview.

[4] CacheLib Cachebench Real Workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval.

[5] Dell 1.6TB, Enterprise, NVMe, Mixed Use Drive, U.2, Gen4 with Carrier. https://www.dell.com/en-us/shop/dell-16tb-enterprise-nvme-mixed-use-drive-u2-gen4-with-carrier/apd/400-bkfk/storage-drives-media#support_section.

[6] Intel Open Cache Acceleration Software. https://open-cas.github.io/.

[7] Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[8] Intel Optane SSD DC P4800X. https://www.intel.com/content/www/us/en/products/sku/97154/intel-optane-ssd-dc-p4800x-series-750gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[9] KIOXIA Enterprise SSDs. https://americas.kioxia.com/en-us/business/ssd/enterprise-ssd.html.

[10] KIOXIA XL-FLASH. https://americas.kioxia.com/en-us/business/memory/xlflash.html.

[11] Linux Block Layer Statistics. https://www.kernel.org/doc/Documentation/block/stat.txt.

[12] Netflix Technology Blog. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef.

[13] NVMe over Fabrics Specification. https://nvmexpress.org/specification/nvme-of-specification/.

[14] Samsung 870 EVO Flash SSD. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/870-evo-sata-2-5-ssd-1tb-mz-77e1t0b-am/.

[15] Samsung 960 EVO Flash SSD. https://semiconductor.samsung.com/consumer-storage/internal-ssd/960evo/.

[16] 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*, ASPLOS '19, page 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 283–300, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *SIGPLAN Not.*, 52(4):631–644, apr 2017.

[19] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 198–204, New York, NY, USA, 2022. Association for Computing Machinery.

[20] 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.

[21] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[22] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, June 1969.

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

[24] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. Nros: Effective replication and sharing in an operating system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 295–312. USENIX Association, July 2021.

[25] Dina Bitton and Jim Gray. Disk shadowing. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, page 331–338, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[26] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 19–31, New York, NY, USA, 1989. Association for Computing Machinery.

[27] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. *Preliminary discussion of the logical design of an electronic computing instrument (1946)*, page 39–48. Ablex Publishing Corp., USA, 1989.

[28] Juneseo Chang, Wanju Doh, Yaebin Moon, Eojin Lee, and Jung Ho Ahn. Idt: Intelligent data placement for multi-tiered main memory with reinforcement learning. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 69–82, 2024.

[29] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. A tale of two paths: Toward a hybrid data plane for efficient Far-Memory applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 77–95, Santa Clara, CA, July 2024. USENIX Association.

[30] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, jun 1994.

[31] Albert Cho and Alexandros Daglis. Starnuma: Mitigating numa challenges with memory pooling. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, MICRO '24, page 997–1012. IEEE Press, 2024.

[32] 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*, MEMSYS '17, page 268–280, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Jonathan Corbet. The future of the page cache. `https://lwn.net/Articles/712467/`, 2017. LWN.net.

[34] Toni Cortes and Jesús Labarta. A case for heterogeneous disk arrays. In *Proceedings IEEE International Conference on Cluster Computing. CLUSTER 2000*, pages 319–325. IEEE, 2000.

[35] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An introduction to the compute express link (CXL) interconnect. *ACM Comput. Surv.*, 56(11), July 2024.

[36] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 381–394, New York, NY, USA, 2013. Association for Computing Machinery.

[37] 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*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

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

[39] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.

[40] Brian C. Forney, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Storage-Aware caching: Revisiting caching for heterogeneous storage systems. In *Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. USENIX Association.

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

[42] Jamel Gafsi and Ernst W. Biersack. Data striping and reliability aspects in distributed video servers. *Cluster Computing*, 2(1):75–91, jan 1999.

[43] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

[44] Binny S Gill. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *FAST*, volume 8, pages 1–17, 2008.

[45] Google. Multichase: A memory latency benchmark. `https://github.com/google/multichase`, 2024.

[46] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[47] Hamid Hadian, Jinshu Liu, Hanchen Xu, Hansen Idden, and Huaicheng Li. Pact: A criticality-first design for tiered memory. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, page 1366–1381, New York, NY, USA, 2026. Association for Computing Machinery.

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

[49] Gangqi Huang, Heiner Litz, and Yuanchao Xu. Pipm: Partial and incremental page migration for multi-host cxl disaggregated shared memory. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, page 1445–1460, New York, NY, USA, 2026. Association for Computing Machinery.

[50] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. Ssds striking back: The storage jungle and its implications on persistent indexes. In *11th Conference on Innovative Data Systems Research, CIDR*, pages 9–12, 2022.

[51] Intel Corporation. Intel Optane Persistent Memory 200 Series. `https://www.intel.com/content/www/us/en/ark/products/series/203877/intel-optane-persistent-memory-200-series.html`, 2020. Accessed: 2026-06-14.

[52] Intel Corporation. Intel Optane SSD DC P5800X Series. `https://www.intel.com/content/www/us/en/products/sku/201860/intel-optane-ssd-dc-p5800x-series-800gb-2-5in-pcie-x4-3d-xpoint/specifications.html`, 2021. Accessed: 2026-06-14.

[53] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, Boston, MA, July 2018. USENIX Association.

[54] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. Limoncello: Prefetchers for scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 577–590, New York, NY, USA, 2024. Association for Computing Machinery.

[55] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[56] Rohan Kadekodi, Haoran Peng, Gilbert Bernstein, Michael D. Ernst, and Baris Kasikci. Jenga: Responsive tiered memory management without thrashing, 2025.

[57] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos — os design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 521–534, 2017.

[58] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. Klocs: kernel-level object contexts for heterogeneous memory systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 65–78, New York, NY, USA, 2021. Association for Computing Machinery.

[59] 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, 2018.

[60] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[61] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[62] K.R. Krish, Ali Anwar, and Ali R. Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 502–511, 2014.

[63] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[64] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[65] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. Association for Computing Machinery.

[66] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 574–587. ACM, 2023.

[67] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1203–1217, New York, NY, USA, 2025. Association for Computing Machinery.

[68] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered memory management beyond hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 731–747, Boston, MA, July 2025. USENIX Association.

[69] Jinshu Liu, Hanchen Xu, Daniel S. Berger, Marcos K. Aguilera, and Huaicheng Li. Performance predictability in heterogeneous memory. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, page 1413–1429, New York, NY, USA, 2026. Association for Computing Machinery.

[70] Mingyu Liu, Li Pan, and Shijun Liu. Rltiering: a cost-driven auto-tiering system for two-tier cloud storage using deep reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems*, 34(2):501–518, 2022.

[71] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S. Vazhkudai. Automatic identification of application I/O signatures from noisy server-side traces. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 213–228, Santa Clara, CA, February 2014. USENIX Association.

[72] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. {eMRC}: Efficient miss ratio approximation for {Multi-Tier} caching. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 293–306, 2021.

[73] 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. *Proc. VLDB Endow.*, 5(10):1076–1087, jun 2012.

[74] Neethu Bal Mallya, Bhavishya Goel, and Ioannis Sourdis. Memplex: A memory system with replication and migration of data for multi-chiplet numa architectures. In *Proceedings of the 39th ACM International Conference on Supercomputing*, ICS '25, page 1219–1233, New York, NY, USA, 2025. Association for Computing Machinery.

[75] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[76] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. Multi-clock: Dynamic tiering for hybrid memory systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*, 2022.

[77] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[78] Sara McAllister, Benjamin Berg, Daniel S Berger, George Amvrosiadis, Nathan Beckmann, Gregory R Ganger, et al. {FairyWREN}: A sustainable cache for emerging {Write-Read-Erase} flash interfaces. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 745–764, 2024.

[79] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[80] Nimrod Megiddo and Dharmendra S Modha. {ARC}: A {Self-Tuning}, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.

[81] Meta. Fbgemm: Facebook general matrix multiplication. `https://github.com/pytorch/FBGEMM`.

[82] Micron Technology. Micron 9300 NVMe SSD (PRO/MAX). https://www.micron.com/products/ssd/product-lines/9300, 2019. Accessed: 2026-06-14.

[83] Micron Technology. DDR5 RDIMM. https://www.micron.com/products/memory/dram-modules/rdimm, 2024. Accessed: 2026-06-14.

[84] Micron Technology. Micron 9550 NVMe Data Center SSD (PRO/MAX). https://www.micron.com/products/storage/ssd/data-center-ssd/9550-ssd, 2024. Accessed: 2026-06-14.

[85] Micron Technology. Micron CZ120 Memory Expansion Module. https://www.micron.com/products/memory/cxl-memory, 2024. CXL 2.0, PCIe Gen5 ×8, 128/256 GB capacities, up to 36 GB/s.

[86] NetApp. How SSD Cache works. docs.netapp.com/us-en/e-series-santricity/sm-storage/how-ssd-cache-works.html, 2023.

[87] NVIDIA. NVIDIA Launches BlueField-4 with STX Storage Architecture, with Broad Industry Adoption. https://nvidianews.nvidia.com/news/nvidia-launches-bluefield-4-stx-storage-architecture-with-broad-industry-adoption, 2025.

[88] 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 *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, February 2012. USENIX Association.

[89] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[90] Nedjah Oussama, Mokhtari Omar, Boumahdi Fatima, and Mancer Yasmine. Autonomous data tiering using reinforcement learning for 3-tier hierarchical storage. In *International Symposium on Modelling and Implementation of Complex Systems*, pages 171–181. Springer, 2024.

[91] B. Ozden, R. Rastogi, and A. Silberschatz. Disk striping in video server environments. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*, pages 580–589, 1996.

[92] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 109–116, New York, NY, USA, 1988. Association for Computing Machinery.

[93] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 557–573, New York, NY, USA, 2023. Association for Computing Machinery.

[94] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. Efficient compactions between storage tiers with prismdb. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 179–193, New York, NY, USA, 2023. Association for Computing Machinery.

[95] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. {EC-Cache}:{Load-Balanced},{Low-Latency} cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, 2016.

[96] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[97] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, page 22–es, New York, NY, USA, 2004. Association for Computing Machinery.

[98] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Mtm: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817, New York, NY, USA, 2024. Association for Computing Machinery.

[99] Yanjing Ren, Yuanming Ren, Xiaolu Li, Yuchong Hu, Jingwei Li, and Patrick PC Lee. {ELECT}: Enabling erasure coding tiering for {LSM-tree-based} storage. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 293–310, 2024.

[100] Yujie Ren, David Domingo, Jian Zhang, Paul John, Rekha Pitchumani, Sanidhya Kashyap, and Sudarsun Kannan. PolyStore: Exploiting combined capabilities of heterogeneous storage. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 539–555, Santa Clara, CA, February 2025. USENIX Association.

[101] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *1986 IEEE Second International Conference on Data Engineering*, pages 336–342, 1986.

[102] Samsung Electronics. Samsung 990 PRO PCIe 4.0 NVMe SSD. `https://www.samsung.com/us/computing/memory-storage/solid-state-drives/990-pro-pcie-4-0-nvme-ssd-2tb-mz-v9p2t0b-am/`, 2022. Accessed: 2026-06-14.

[103] Samsung Electronics. Samsung CMM-D (CXL Memory Module-DRAM). `https://semiconductor.samsung.com/cxl-memory/cmm-d/`, 2024. CXL 2.0, PCIe 5.0, up to 256 GB DDR5 per module.

[104] Samsung Semiconductor. Samsung PM893 Data Center SATA SSD. `https://semiconductor.samsung.com/ssd/datacenter-ssd/pm893/`, 2023. Accessed: 2026-06-14.

[105] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. Vtmm: Tiered memory management for virtual machines. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 283–297, New York, NY, USA, 2023. Association for Computing Machinery.

[106] SK hynix. SK hynix CMM-DDR5 CXL Memory Module. `https://product.skhynix.com/products/cmm/cmm.go`, 2022. 96 GB EDSFF E3.S, PCIe 5.0 ×8, 24 Gb DDR5 on 1a-nm process; first samples Q4 2022.

[107] Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. Hybridtier: an adaptive and lightweight cxl-memory tiering system. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '25, page 112–128, New York, NY, USA, 2025. Association for Computing Machinery.

[108] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Prism: Optimizing key-value store for modern heterogeneous storage devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 588–602, New York, NY, USA, 2023. Association for Computing Machinery.

[109] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[110] Wei Su, Abhishek Dhanotia, Carlos Torres, Jayneel Gandhi, Neha Gholkar, Shobhit Kanaujia, Maxim Naumov, Kalyan Subramanian, Valentin Andrei, Yifan Yuan, and Chunqiang Tang. Dcperf: An open-source, battle-tested performance benchmark suite for datacenter workloads. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1717–1730, New York, NY, USA, 2025. Association for Computing Machinery.

[111] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. {RIPQ}: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[112] The Linux Kernel Project. Concepts Overview: The Linux Kernel Memory Management Documentation. `https://docs.kernel.org/admin-guide/mm/concepts.html`, 2024. Linux kernel admin-guide.

[113] John Tramm, Andrew Siegel, Tanzima Islam, and Martin Schulz. Xsbench - the development and verification of a performance abstraction for monte carlo reactor analysis. 09 2014.

[114] Kaiwei Tu, Kan Wu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Getting the MOST out of your storage hierarchy with Mirror-Optimized storage tiering. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, pages 561–578, Santa Clara, CA, February 2026. USENIX Association.

[115] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[116] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 79–94, New York, NY, USA, 2024. Association for Computing Machinery.

[117] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the host network. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 581–594, New York, NY, USA, 2024. Association for Computing Machinery.

[118] Hui Wang and Peter Varman. Balancing fairness and Efficiency in tiered storage systems with Bottleneck-Aware allocation. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 229–242, Santa Clara, CA, February 2014. USENIX Association.

[119] Western Digital. Ultrastar DC HC690 Data Center Hard Drive. `https://www.westerndigital.com/products/internal-drives/data-center-drives/ultrastar-dc-hc690-hdd`, 2024. Accessed: 2026-06-14.

[120] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. *ACM Trans. Comput. Syst.*, 14(1):108–136, feb 1996.

[121] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, feb 2019.

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

[123] 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. USENIX Association, February 2021.

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

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

[126] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-Exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, Santa Clara, CA, July 2024. USENIX Association.

[127] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive page profiling and migration for tiered memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 817–833, Santa Clara, CA, July 2024. USENIX Association.

[128] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

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

[130] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jinyong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, Boston, MA, July 2023. USENIX Association.

[131] 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.

[132] Xinyue Yi, Hongchao Du, Yu Wang, Jie Zhang, Qiao Li, and Chun Jason Xue. Artmem: Adaptive migration in reinforcement learning-enabled tiered memory. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 405–418, 2025.

[133] Dongha Yoon, Hansen Idden, Jinshu Liu, Berkay Inceisci, Sam H. Noh, and Huaicheng Li. Cylon: Fast and accurate Full-System emulation of CXL-SSDs. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, pages 313–327, Santa Clara, CA, February 2026. USENIX Association.

[134] Jianping Zeng, Shuyi Pei, Da Zhang, Yuchen Zhou, Amir Beygi, Xuebin Yao, Ramdas Kachare, Tong Zhang, Zongwang Li, Marie Nguyen, Rekha Pitchumani, Yang Soek Ki, and Changhee Jung. Performance characterizations and usage guidelines of samsung cxl memory module hybrid prototype, 2025.

[135] 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, 2010.

[136] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu. What's the story in EBS glory: Evolutions and lessons in building cloud block store. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 277–291, Santa Clara, CA, February 2024. USENIX Association.

[137] 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, Boston, MA, February 2019. USENIX Association.

[138] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024. USENIX Association.

[139] 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*, SIGMOD '21, page 2195–2207, New York, NY, USA, 2021. Association for Computing Machinery.