

Architecting Memory Efficiency in Modern Data Centers

Kaiyang Zhao

CMU-CS-26-121

July 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Dimitrios Skarlatos, Chair
Phillip Gibbons
Todd Mowry
Kim Keeton, Google

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2026 Kaiyang Zhao

This research was sponsored by: Intel Corporation Santa Clara under award numbers 1012901 and 1012513; the National Science Foundation under award number CNS2239311; Qualcomm Fellowship under award number 5009543; and the Scott Institute Seed Award under award number 5005257. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: memory, virtual memory, address translation, page tables, memory tiering, memory offloading, multi-tenancy

Abstract

Over the past few decades, a confluence of factors including i) the proliferation of memory-intensive applications, ii) the rapid expansion of memory capacity towards the terabyte scale, and iii) the slowing of DRAM technology improvements, has made memory scaling difficult and established memory as the critical bottleneck in modern data center computing. The memory bottleneck manifests in multiple dimensions, including access performance, hardware costs, and power consumption.

First, the virtual memory abstraction is under increasing strain. As memory capacity grows while the Translation Lookaside Buffer (TLB) size remains stagnant, virtual address translation overhead becomes more severe; internal profiling at hyperscalers reveals that approximately 20% of CPU cycles are performing page walks due to TLB misses. Furthermore, this overhead will worsen due to the inherent limits of TLB size scaling, the introduction of additional page table levels, hardware virtualization for VMs, the vast memory capacity enabled by heterogeneous memory, and the reliance on the page-level security checks of confidential computing. Without more efficient virtual memory designs, memory will be too slow to scale.

Second, the hardware costs and power consumption of memory have skyrocketed. DRAM now accounts for nearly a quarter of rack power consumption and nearly half of the total cost of ownership (TCO) of a typical data center server. Looking forward, these costs will continue to escalate due to the increasing demands of memory-intensive applications. Such high costs hinder the data center build-out essential for supporting the insatiable demand of AI applications, and increase the operating costs of existing data centers.

This thesis addresses the multifaceted memory efficiency challenge through the co-design of operating systems (OS) and computer architecture.

To improve memory access performance, this thesis presents two works that reduce address translation overhead. **Contiguitas** creates abundant physical memory contiguity by grouping unmovable allocations in the OS and introducing hardware extensions to migrate previously unmovable pages used for device I/O. Contiguitas then utilizes the newly found physical memory contiguity to allocate more huge pages for applications and reduce the address translation overhead, yielding up to 18% performance improvement for Meta’s production workloads. More importantly, the large amount of physical memory contiguity provided by Contiguitas sets up a solid foundation for future work on address translation. Second, Learned Virtual Memory (**LVM**) replaces the rigid structure of radix page tables with learned page table structures that are tailored to applications’ virtual address spaces. LVM leverages the regularity of application virtual address spaces to build learned indexes for address translation, and balances the size, depth, and accuracy of the learned indexes through a cost model tailored for address translation. LVM enables low-cost virtual address space changes, supports variable page sizes, and is affordable in the hardware.

LVM reduces page walk overhead by an average of 44% for a set of memory-intensive applications and achieves a 2–27% speedup in execution.

To reduce the cost and power consumption of memory, this thesis presents two works that enable the practical deployment of CXL memory in modern data centers. Squeezed by the high cost of DRAM and the need for memory capacity, data center operators have explored replacing part of memory with cheaper but slower CXL memory backed by reused last-generation DRAM, along with SSD swap, as offloading tiers. **Pensieve** introduces a three-tier memory system to transparently manage pages across DRAM, CXL memory, and SSDs, and aims to maximize DRAM savings by utilizing cheaper memory tiers while maintaining application performance. It manages the system through a single-input, single-output control loop, is robust against workload access pattern changes, and leverages CXL memory as an intermediate tier to observe the hotness trends of pages. Pensieve overcomes the capacity limitations of CXL memory and the performance limitations of SSD swap. Pensieve safely offloads 33% of memory on average, substantially outperforming standalone DRAM+CXL (21%) and DRAM+SSD (18%) setups, while keeping application performance degradation below 5%. **Equilibria** addresses the challenges of deploying tiered memory in today’s multi-tenant environments. It ensures fair memory sharing across memory tiers and mitigates noisy-neighbor effects through automatic fair-share determination, flexible fair-share policies, and thrashing mitigation. Equilibria consistently helps workloads meet their service-level objectives (SLOs) while avoiding performance interference, improving performance over the state-of-the-art Linux solution by up to 52% and $1.7\times$ for Meta’s production workloads and open-source benchmarks, respectively.

Together, these works show that through a comprehensive set of novel architectural and operating system techniques that work together, the memory efficiency in modern data centers can be significantly improved from the perspectives of performance, hardware costs, and power. This thesis lays the foundation for memory systems that can scale with evolving application demands and hardware trends, while opening a broad set of directions for future research in computer architecture and operating systems. Several of the works presented in this thesis have already been upstreamed into Linux and deployed across production data centers on millions of servers, helping scale memory systems for years to come.

Acknowledgments

I am very fortunate to have interacted with many wonderful people during my PhD journey, and I want to take this opportunity to sincerely thank them.

First and foremost, I would like to thank my advisor, Professor Dimitrios Skarlatos. He provided timely and insightful guidance every step of the way and repeatedly showed me what a true passion for research looks like. I have learned so much from him—from selecting compelling research topics and appreciating the importance of high-quality writing and presentation, to having the infinite optimism needed to power through the highs and lows of research.

I am also grateful to my thesis committee members, Professors Phillip Gibbons and Todd Mowry, for their valuable feedback on this thesis and my overall research direction. I would also like to thank my external committee member, Kim Keeton, who braved the time zone differences and always offered thoughtful observations and encouragement.

Furthermore, I would like to thank my collaborators at Meta—Dan, Leon, Antonis, Johannes, Bikash, CQ, Neha, Abhishek, and Hasan—who introduced me to industry research at a hyperscaler. I would also like to thank my collaborators at Google, Suli and Kim, who taught me how to conduct rigorous and principled research in an industry setting. I also want to thank my undergraduate and graduate mentees, Chloe and Xenia, who took on crucial experiments and provided me with fresh perspectives on research.

Next, I want to thank all my labmates in the CAOS group: Sid, Patrick, Eliot, and Vasilis, as well as my officemate, Matias. Research would have been so much less fun without them. I am especially grateful to my friends Ke, Mingkuan, Ziyue, Yuang, and Yue for our weekly dinners and random conversations that were simultaneously hilarious and profound. They are the behind-the-scenes heroes who helped me power through this journey.

Going back to the beginning of my academic path, I am thankful to Professors Pedro Fonseca and Felix Lin at Purdue University, who took a chance on a clueless undergraduate student dipping his toe into research.

Finally, I am deeply grateful to my parents, who allowed me to spend countless hours on computers and develop my interest in them. They have backed my choices with love and unconditional support, and made this journey possible.

Contents

1	Introduction	1
2	Contiguitas: Physical Memory Contiguity and Huge Pages	5
2.1	Introduction	5
2.2	Memory Contiguity Challenges and Opportunities in Data Centers	8
2.2.1	Memory Management	8
2.2.2	Memory Capacity and TLB Trends	9
2.2.3	Lost Cycles Due to Lack of Contiguity	10
2.2.4	Memory Fragmentation	11
2.2.5	Unmovable Memory Allocations	12
2.3	Contiguitas Design	14
2.3.1	Overview	14
2.3.2	Confining Unmovable Allocations	15
2.3.3	Architectural Support for Transparent Page Mobility	17
2.4	Evaluation Methodology	22
2.5	Evaluation	23
2.5.1	End-to-end Performance Impact	23
2.5.2	Unmovable Allocations and Memory Contiguity.	24
2.5.3	Contiguitas-HW Characterization.	26
2.6	Discussion	28
2.7	Related Work	28
2.8	Conclusion	29
3	Learned Virtual Memory: A Learned Page Table Design	31
3.1	Introduction	31
3.2	Background	33
3.2.1	Radix Page Tables	33
3.2.2	Hashed Page Tables	34
3.2.3	Learned Indexes	35
3.3	Learned Indexes for Page Tables	35
3.3.1	Virtual Address Space Regularity	35
3.3.2	Virtual Memory Challenges	36
3.4	Learned Virtual Memory Design	38
3.4.1	Overview	38

3.4.2	A Learned Index for Virtual Memory	40
3.4.3	Learned Index Operations	43
3.4.4	Supporting Multiple Page Sizes	45
3.4.5	Fixed-Point Arithmetic for Fast Lookups	46
3.4.6	OS and Hardware Support	47
3.5	Implementation	48
3.5.1	Parameters Used	48
3.5.2	LVM Operations in the OS	49
3.5.3	Linux Prototype	50
3.6	Evaluation Methodology	50
3.6.1	Architectural Simulation	50
3.6.2	Workloads	50
3.6.3	Configurations and Baselines	50
3.7	Evaluation	51
3.7.1	End-to-end Speedups.	51
3.7.2	Architectural Characterizations.	52
3.7.3	Learned Page Table Characterization	55
3.7.4	Hardware Characterization	57
3.7.5	Additional Comparison to Prior Work	57
3.8	Related Work	58
3.9	Discussion and Future Work	59
3.10	Conclusion	59
4	Pensieve: Towards Multi-Tier Memory Systems	61
4.1	Introduction	61
4.2	Background	63
4.2.1	Swap-Based Two-Tier Systems	63
4.2.2	Byte-Addressable Two-Tier Systems	64
4.3	Requirements and Challenges	64
4.3.1	Problem Formulation	65
4.3.2	Challenges	66
4.4	Design	67
4.4.1	System Overview	67
4.4.2	High-Level Design Decisions	68
4.4.3	Indicator Metrics	71
4.4.4	Control Algorithm	72
4.4.5	Static Data Path Configuration	73
4.5	Implementation	74
4.5.1	Missing Linux Support for Three-Tier Memory	75
4.5.2	Flexible Coldness Tracking	75
4.6	Evaluation	75
4.6.1	DRAM Savings and Performance	76
4.6.2	System Stability	78
4.6.3	Validating Design Decisions	80

4.7	Related Work	81
4.8	Conclusion	82
5	Equilibria: Fair and Multi-Tenant CXL Memory Tiering At Scale	83
5.1	Introduction	83
5.2	Background	85
5.2.1	Memory Costs in Data Centers and CXL Memory	85
5.2.2	Containers for Multi-Tenancy	86
5.3	Learnings from Production	86
5.3.1	Real vs. Emulated CXL Memory	86
5.3.2	Capacity-Bound Apps Use Little CXL Bandwidth	87
5.3.3	CXL Does Not Remove Bandwidth Bottlenecks	87
5.3.4	Most Applications Require Transparent Tiering	88
5.3.5	Hardware Support for Tiering is Insufficient	88
5.3.6	Multi-Tenancy and Fairness	89
5.4	Equilibria Design	90
5.4.1	Fairness: Definition and Challenges	90
5.4.2	System Overview	91
5.4.3	Automatic Determination of Fair Share	92
5.4.4	Container-level Observability and Control	94
5.4.5	Modulating Demotions	94
5.4.6	Regulating Promotion	95
5.4.7	Mitigating Thrashing	96
5.4.8	Implementation	97
5.5	Evaluation	97
5.5.1	Methodology	97
5.5.2	Validating Tiering Control	98
5.5.3	Characterizing Fair Share Determination	100
5.5.4	Evaluating Equilibria with Benchmarks	101
5.5.5	Deploying Equilibria in Production	102
5.6	Related Work and Discussions	105
5.7	Conclusion	106
6	Conclusion	107
6.1	Future Work	108
	Bibliography	109

List of Figures

2.1	Page migration procedure.	9
2.2	Memory and TLB coverage of computing hardware across generations. . .	10
2.3	Percentage of cycles lost due to page walks.	11
2.4	Contiguity availability as the percentage of free memory.	12
2.5	Distribution of unmovable pages in contiguous regions of 2MB, 4MB, 32MB, and 1GB.	13
2.6	Sources of unmovable allocations.	13
2.7	Contiguitas design overview.	14
2.8	Contiguitas hardware overview. (a) shows the hardware extension of Contiguitas-HW in the LLC. (b) shows the metadata table of Contiguitas-HW. (c) shows the page migration process followed by Contiguitas-HW.	18
2.9	Contiguitas operations in a sliced last-level cache architecture.	20
2.10	End-to-end performance over Meta’s production workloads. The red bar of Contiguitas for Web shows the performance gains from 1GB pages.	24
2.11	Unmovable 2 MB pages for Meta’s production workloads.	25
2.12	Potential memory contiguity as a percentage of total memory.	26
2.13	Page unavailable cycles during migration.	27
3.1	Address translations schemes. LVM avoids both the long sequential page walks of radix page tables and the high collision rates and parallel accesses of hashed page tables.	34
3.2	Virtual memory gap coverage of gap = 1.	35
3.3	Median percentage of free memory in a Meta’s data center that can be allocated contiguously at various sizes.	37
3.4	LVM learns the distribution of virtual addresses and maps them to page tables.	37
3.5	LVM out of bounds inserts close to the edge.	45
3.6	Regular and huge pages as represented by LVM.	46
3.7	LVM hardware overview.	47
3.8	LVM Page Walk Cache Entry.	48
3.9	End-to-end speedups of various page table designs.	52
3.10	MMU overhead relative to radix. Results are normalized separately to radix 4KB and THP.	53
3.11	Page walk traffic relative to radix. Results are normalized separately to Radix 4KB and THP.	54

3.12	Cache MPKI relative to radix page tables.	54
4.1	Pensieve high-level design. Black arrows mark data path operations. The control paths are shown in green.	68
4.2	Pensieve system overview.	74
4.3	DRAM tier savings and performance losses measured by application-specific metrics of two-tier systems and Pensieve. To the right of the dotted line is a Meta production workload. RB does not report performance.	78
4.4	DRAM savings at various application-reported performance losses, comparing Pensieve and two-tier systems using memcached (top) and mediawiki (bottom). The larger dots show the operating points of Figure 4.3.	79
4.5	Timeseries of memory usage of memcached.	80
4.6	Timeseries of STAR and PSI of memcached.	80
4.7	Timeseries of active constraints of memcached.	81
5.1	Memory as a % of rack TCO and power at Meta.	86
5.2	Latency versus bandwidth for different memory tiers.	87
5.3	Unfairness in existing tiered memory systems. Container A with hotter access patterns gets 100% local memory, whereas Container B gets only about half.	89
5.4	Equilibria overview.	91
5.5	Memory usage when containers exceed local memory lower protection. . . .	98
5.6	Promotion activities when containers exceed local memory lower protection. Container A's promotion rate (blue line) is throttled.	99
5.7	Memory placement of TaoBench and SparkBench (Container A) on baseline Linux. Container A is forced to use increasingly more CXL memory and less local memory.	101
5.8	Memory usage on Equilibria of four CI instances.	103
5.9	Memory usage on Equilibria of five instances of Web.	104

List of Tables

2.1	Contiguitas architectural parameters.	23
3.1	LVM architectural parameters.	51
3.2	Index size of LVM in bytes.	55
4.1	Summary of evaluated applications in Pensieve.	76
5.1	Bandwidth usage of memory capacity-bound apps.	87
5.2	Bandwidth and capacity scaling of memory.	88
5.3	Application performance gains from CXL bandwidth.	88

Chapter 1

Introduction

Over the past few decades, a confluence of factors has reshaped the landscape of memory systems and created new memory efficiency challenges. First, the proliferation of memory-intensive applications has led to increased memory usage. Second, the memory capacity of servers in modern data centers has expanded in response, scaling from gigabytes to terabytes. Finally, the slowing of DRAM technology improvements compared to other hardware components means DRAM is taking up an increasingly larger share of hardware costs and power consumption. As a result, memory has become a critical bottleneck in modern data center computing. The memory bottleneck manifests in multiple dimensions, including access performance, hardware costs, and power consumption. To maintain efficiency in data center computing while continuing to scale memory, memory efficiency must be holistically and significantly improved.

The High Overhead of Virtual Memory. Virtual memory is the cornerstone abstraction that applications use to address memory, offering essential benefits such as isolation, protection, programmability, and transparent data placement. However, virtual memory requires an address translation process during every memory access, adding overhead to application execution. Modern operating systems have largely stuck with a small 4KB base page size, and as memory capacities grow, virtual memory implementations are plagued by expansive page table trees and a corresponding appetite for hardware Translation Lookaside Buffer (TLB) capacity that is difficult to satiate. In particular, trends in data center hardware show increases in memory capacity without comparable increases in TLB capacity in the past decade: memory capacity has grown from dozens of gigabytes to hundreds of gigabytes or even a few terabytes, yet the TLB size has stagnated at thousands of entries. As a result, TLB coverage is woefully insufficient with contemporary 4 KB pages, and even larger 2 MB pages are not a panacea.

Applications are already suffering a substantial performance penalty due to the mismatch between memory capacity and TLB coverage. For example, our study shows that the percentage of page walker active cycles can account for close to 20% of total cycles for production workloads in Meta’s data centers [235], with similar results also reported by Google [85]. This virtual memory overhead reduces the performance of applications, ultimately reducing the fleet-wide efficiency of data centers and making continued scaling of memory capacity unsustainable.

Unfortunately, this overhead is only bound to worsen due to several impending architectural and application trends: i) the inherent hardware limits of TLB scaling, which have pushed latencies beyond those of L2 caches [221], ii) the advent of terabyte-scale memory capacity through technologies like CXL [91, 182], iii) the introduction of additional levels of page tables (e.g., 5-level paging [87]), iv) the proliferation of hardware virtualization for VMs, v) the continuing increase of memory-intensive applications, and vi) confidential computing platforms that place page-granularity security checks on the critical path during address translation [17, 89]. To keep scaling the memory capacity and adding new memory features without wasting an increasingly larger share of cycles on virtual memory, we must bring the virtual memory abstraction to the land of efficiency.

The Escalating Cost and Power Consumption of DRAM. As of the early 2020s, DRAM comprises 40–50% of server costs in data centers [150, 216], and is one of the biggest contributors towards rack-level capital expenditures and operating expenses [123, 145]. More recently, the trend toward increasing DRAM cost and power contributions has intensified, with DRAM prices doubling due to the immense memory demands of AI applications [135] and data centers being constrained by the power grid capacity. DRAM now accounts for nearly a quarter of rack power consumption and up to half of the total cost of ownership (TCO) of a typical server in modern data centers [237]. The reasons for this trend are twofold. First, workloads are dealing with more data, hence memory bandwidth and capacity requirements per unit of work have grown over time; servers must therefore have more DRAM installed. Second, memory technologies have not been scaling at the same rate as the other hardware components; DRAM’s share of server cost and power grows as a result. To make matters worse, DRAM is not only more expensive but also more scarce, as DRAM demand far outstrips supply [135].

To rein in the ballooning cost and power of memory, data center operators have looked towards Compute Express Link (CXL) [46]. CXL is a cache-coherent interconnect that enables various memory technologies, including DDR, LPDDR, NAND, and persistent memory, to be connected to the host server. It offers greater flexibility in terms of memory capacity, cost, power, and performance characteristics. Hyperscalers are deploying reused older-generation DRAM as *CXL memory* that provides part of a server’s memory capacity at very low costs or expands a server’s memory capacity, therefore reducing the overall spending on memory at the data center level; LPDDR-backed CXL memory can optimize the power profile of servers. However, CXL memory has increased access latency compared to CPU-attached DRAM, which can negatively impact application performance when significant portions of memory accesses are to CXL memory. This means that software support is required to effectively use CXL memory without compromising application performance. Many hyperscalers have embraced tiered memory systems, where data that is less likely to be accessed or less performance-critical is demoted from expensive main memory to CXL memory. The problem is that existing tiered memory systems cannot be integrated with multi-tenancy and multi-memory tier setups in modern data centers. To affordably scale memory capacity for the future, we must overcome these limitations and enable the practical deployment of CXL memory in data centers.

The goal of this thesis is to understand the inefficiencies present in current memory systems, identify the root causes of such inefficiencies, and design new computer architectures

and operating systems that improve memory efficiency and enable the continued scaling of memory capacity sustainably and affordably. The contributions of this thesis are two-part, from the perspectives of both i) virtual memory overhead and ii) memory hardware and power costs.

First, this thesis presents two works that drastically reduce the address translation overhead by mitigating virtual memory inefficiencies caused by the lack of physical memory contiguity and the rigid page table structure.

Contiguitas (ISCA’23) [235] (Best Paper Award, MICRO Top Picks 2024, Upstreamed to Linux) addresses the extreme scarcity of physical memory contiguity in data centers. Through a fleet-wide study at Meta, we found that 23% of servers do not have physical memory contiguity for even a single 2MB huge page, and that this fragmentation happens rapidly, primarily driven by unmovable memory allocations (such as networking buffers, slab caches, and page tables). Contiguitas proposes software techniques in the operating system to separate movable allocations from unmovable ones into two dynamically adjusted continuous regions. For the majority of unmovable allocations that are tied to device I/O (e.g., networking), Contiguitas introduces targeted hardware extensions in the last-level cache (LLC) that enable the transparent migration of unmovable pages while in use. This relies on migration mappings in the LLC and a relaxed TLB shutdown operation.

Impact: Contiguitas successfully confines unmovable allocations and applies the newly abundant memory contiguity to reduce the virtual memory translation overhead, achieving up to 18% performance improvement for Meta’s production applications and enabling the dynamic allocation of 1GB huge pages for more performance gains. Through a collaboration with Meta and Linux kernel maintainers, the core ideas of Contiguitas have been upstreamed into Linux and are now running in production.

Learned Virtual Memory (**LVM**) (MICRO’25) [236] (MICRO Top Picks 2025 Honorable Mention) replaces the rigid structure of radix page tables with learned indexes tailored to the specific characteristics of an application’s virtual address space. Recognizing that application virtual address spaces exhibit significant regularity, LVM leverages small yet effective linear models to enable single-access address translation. To overcome the limitations of traditional learned indexes, LVM introduces a cost model that balances prediction accuracy with index cacheability in the MMU. LVM enables efficient, dynamic updates of the virtual address space by leveraging the contiguous expansion of virtual memory to scale linear models without retraining. Furthermore, it uses adaptive learned indexes that allocate small leaf page tables based on the widely available physical contiguity of hundreds of kilobytes, removing the need for large contiguous allocations. Finally, it supports multiple page sizes through a novel encoding scheme using varying slopes in the linear models.

Impact: LVM reduces the page walk overhead by an average of 44% over radix page tables, shrinks the page walk cache area by $1.5\times$, and achieves a 2–27% speedup in application execution.

The second part of this thesis introduces operating system support for emerging tiered-memory hardware, addressing the challenges of managing multiple memory tiers under multi-tenancy and enabling practical, memory-efficient deployment in modern data centers.

Pensieve introduces a three-tier memory system that transparently manages pages across DRAM, CXL memory, and SSDs to maximize DRAM savings while maintaining

application performance. To navigate the complex multi-dimensional decision space, Pensieve combines a hierarchical data plane, decoupled control loops, and unified feedback-driven control. The hierarchical data plane treats DRAM as hot, CXL as warm, and SSD as cold, using CXL as an intermediate filter to preserve SSD endurance and maintain performance. Pensieve decouples the management into a DRAM-CXL control loop and a CXL-SSD control loop, significantly simplifying the tight coupling of the tiers. A unified feedback-driven control mechanism dynamically adjusts the tiering aggressiveness based on estimated performance loss and tier bandwidth utilization.

Impact: Pensieve safely offloads 33% of memory on average, substantially outperforming standalone DRAM+CXL (21%) and DRAM+SSD (18%) setups, while overcoming the capacity limitations of CXL memory and the performance limitations of SSD swap and observing the system constraints. Pensieve has been incorporated as part of the software support for CXL memory at Google.

Equilibria addresses the fairness challenges of deploying CXL memory in today’s multi-tenant data center environments. Without fairness, colocated workloads suffer severe performance anomalies and service-level objective (SLO) violations due to noisy-neighbor effects and imbalanced memory tier usage. Equilibria is a comprehensive operating system solution that provides multi-tenancy-aware controls and observability for tiered memory. It features automatic fair-share determination, which online-profiles and solves an optimization problem to allocate CXL memory capacity, access, and bandwidth fairly among scheduled jobs. Equilibria enforces these policies through regulated promotion and demotion, and includes dedicated thrashing mitigation to protect the system from disruptive workloads.

Impact: Equilibria consistently helps workloads meet their SLOs, avoiding performance interference, and improves performance over the state-of-the-art Linux solution by up to 52% and $1.7\times$ for Meta’s production workloads and open-source benchmarks, respectively. The core designs of Equilibria have been upstreamed to Linux kernel and deployed in production to support CXL memory-equipped servers at Meta.

These works encompass a comprehensive set of novel architectural and operating system techniques that work together and improve the memory efficiency in modern data centers from the perspectives of performance, hardware costs, and power, addressing the urgent need for efficiency in data centers. This thesis lays the foundation for memory systems that can scale with evolving application demands and hardware trends, while opening a broad set of directions for future research in computer architecture and operating systems. Several of the works presented in this thesis have already been upstreamed into Linux and deployed across production data centers on millions of servers, helping scale memory systems for years to come.

Chapter 2

Contiguity: Physical Memory Contiguity and Huge Pages

2.1 Introduction

Memory capacity has increased dramatically over the last decades, yet modern operating systems have throughout stuck with a small base page size. This divergence has resulted in excessive management overhead for memory-intensive applications. In particular, virtual memory implementations are plagued by expansive page table trees, and a corresponding appetite for hardware TLB capacity that is difficult to satiate.

Even with architectural innovations such as larger and multi-level TLBs and page walk caches [7, 23, 24, 25, 31, 32, 33, 34, 48, 107, 160], applications today suffer a substantial performance penalty due to TLB misses. Google’s internal profiling revealed that approximately 20% of cycles are stalled on TLB misses [85]. Unfortunately, this problem is only bound to get worse due to: i) the inherent hardware limits of TLB scaling, already surpassing L2 cache latencies [221], ii) terabyte-scale memory capacity through technologies like CXL [91, 182], iii) additional levels of page tables [87], iv) the increase of memory-intensive applications, and v) upcoming confidential computing platforms that place security checks at page granularity during address translation [17, 89].

A large body of prior research has focused on reducing the address translation overhead [1, 3, 4, 23, 25, 30, 34, 50, 63, 79, 85, 107, 111, 142, 156, 162, 180, 188, 190, 193, 194, 198, 210, 228]. Conceptually, we can separate prior work based on the amount of physical memory contiguity required and how it is exploited. On the one hand, earlier works propose leveraging physical memory contiguity to back the application dataset [7, 25, 63, 64, 106, 107, 160]. These approaches create range-based translations, similar to segments, that map large contiguous regions of virtual memory to equally large contiguous physical memory. Their goal is to ultimately reduce the number of TLB entries needed. However, they face the fundamental challenge that it is very hard to create multi-gigabyte contiguous physical address ranges to cover the complete application dataset.

On the other hand, another line of research has explored alternative page table structures such as hashed page tables [54, 84, 97, 100, 188, 193, 197, 228]. Indeed, recent

work [188, 193, 194, 228] has solved some of their traditional shortcomings [24, 56, 70, 86, 95]. Such solutions aim to replace sequential multi-level page tables and drastically reduce the cost of page walks by accelerating page table accesses. Notably, they relax the physical memory contiguity requirements to apply not on the whole dataset, but only on the page table organization. However, they impose strict requirements for physical memory contiguity availability on the critical path of page table creation. As a result, such approaches still remain challenging to adopt. Several other architectural extensions that implicitly rely on contiguity [75, 143, 160, 210, 225] are hindered by the same fundamental challenge.

Today’s operating systems, such as Linux, have mostly relied on 2 MB huge pages to land performance improvements. The primary mechanism to leverage huge pages is Transparent Huge Pages (THP) [204] that *opportunistically* try to provide 2 MB pages. Unfortunately, in today’s systems, finding physical contiguity even for 2 MB pages is often hard due to memory fragmentation [65, 68, 74, 117, 155, 156, 162]. THPs have also been under scrutiny due to their performance implications such as latency spikes and memory bloating [15, 38, 44, 65, 74, 130, 149]. Alternative approaches, such as userspace allocators [85, 136, 137] still rely on the OS to provide physical contiguity and larger mappings.

Addressing excessive memory management overhead will require a fundamental shift towards larger page granularities. However, fragmentation remains as the primary obstacle. Illuminator [155] is a major prior work that tries to address fragmentation. Within a 2 MB block, it prevents mixing *unmovable* allocations and *movable* ones that the kernel can move on-demand. This work, while innovative, has some key limitations.

First, the fundamental problem of unmovable allocations, i.e., that the OS cannot move them after their allocation, remains. Second, avoiding mixing movable and unmovable allocations within 2 MB blocks still fragments the address space, prevents large contiguous regions from being formed, and consequently blocks larger translations. This is because it limits the maximum available contiguity to at most 2 MB. For example, a single unmovable 4 KB page can render a 1 GB region unmovable; as a result just 0.19% of 2 MB unmovable allocations can fragment the whole of memory irrecoverably. Third, the effectiveness of defragmentation is severely hindered by the presence of unmovable allocations [225]. Overall, despite significant efforts in virtual memory research, eliminating the address translation overhead will remain onerous in real world environments until the physical memory contiguity predicament gets resolved.

In this work, we start with a detailed investigation of physical memory contiguity at hyperscale across Meta’s data centers. We sample servers across the fleet and show that *23% of servers do not even have physical memory contiguity for a single 2 MB huge page*. We also find that it is practically impossible to dynamically allocate 1 GB pages in a production environment. Pertinently, fragmentation affects all servers as there is little to no correlation between memory contiguity availability and server up-time. In practice, servers can quickly get heavily fragmented within the first hour after boot-up while the mean server uptime is multiple days or weeks—turning memory fragmentation into a major challenge. Finally, our study exposes unmovable memory allocations as the root cause for the lack of physical memory contiguity. In particular, we identify several sources of unmovable allocations, including networking buffers, slab, filesystem, and page tables.

To address these issues, we introduce Contiguitas with the goal of eliminating fragmentation due to unmovable allocations. Contiguitas separates movable allocations from unmovable ones by placing them into two different continuous regions and dynamically adjusts the boundary of the two regions based on memory demand. To avoid wasting memory in the unmovable region, Contiguitas solves two problems: i) how to dynamically resize the unmovable region and place unmovable allocations; and ii) how to drastically reduce unmovable allocations. For the first problem, Contiguitas performs resizing by tracking the demand for unmovable allocations. Moreover, it reduces internal fragmentation of the unmovable region by differentiating different types of unmovable allocations.

For the second problem, Contiguitas focuses on unmovable allocations that cannot be moved with software alone because access to the page cannot be blocked for a migration to take place. At Meta, networking allocations account for 73% of unmovable pages (Section 2.2.5). We expect unmovable allocation to become an increasingly bigger problem. This is because of new I/O technologies such as kernel-bypass and RDMA for networking and storage, GPUs, and other accelerators that heavily rely on unmovable pages.

To this end, Contiguitas introduces a set of surgical hardware extensions in the last-level cache (LLC) that enable the *transparent* migration of unmovable pages while in use. Contiguitas’s design builds off of two ideas: First, Contiguitas introduces migration mappings in the LLC, enabling hardware to redirect traffic to the appropriate cache line of each page based on the progress of the migration. Second, Contiguitas relaxes the TLB shutdown operation from being synchronous and requiring acknowledgments from all victim TLBs to a local TLB invalidation that can be performed by each core independently and in a lazy manner. Naturally, movable page migrations can also benefit from this hardware support.

We build the OS component of Contiguitas into the Linux kernel and run our experiments in Meta’s production environment. Our results show that this component successfully confines unmovable allocations, drastically reducing unmovable 2MB blocks from an average of 31% scattered across the address space down to 7% confined in the unmovable region, leading to significant performance gains. Specifically, we show that for three major production services, Contiguitas achieves performance improvements between 2-9% for partially fragmented servers that represent the majority of the servers, and between 7-18% for highly fragmented servers representative of nearly a quarter of the fleet at Meta. Notably, Contiguitas’s contiguity gains enable Web, one of Meta’s largest services, to dynamically allocate 1 GB huge pages, leading to a 7.5% performance win that is unattainable with 2 MB pages alone. We use full-system simulations to demonstrate the effectiveness of the hardware extensions of Contiguitas. Our evaluation shows that Contiguitas-HW enables the efficient migration of unmovable allocations while scaling the number of victim TLBs and does not affect application performance. We are currently in the process of upstreaming Contiguitas into Linux.

2.2 Memory Contiguity Challenges and Opportunities in Data Centers

In this section, we first provide a brief overview of memory management of modern operating systems. Then we showcase the challenges and opportunities of memory contiguity across Meta’s data centers through a detailed study of i) memory capacity and TLB trends, ii) performance implications of address translation, iii) memory fragmentation, and iv) unmovable allocations and their sources.

2.2.1 Memory Management

Multiple Page Sizes. Modern operating systems, such as Linux, manage memory in small 4 KB page granularity. The primary reason behind this decision is to reduce memory bloating and expensive IO operations during paging. However, 4 KB pages cause significant address translation overhead due to the limited capacity of the TLBs and expensive page walks in the event of a TLB miss. To that end, huge pages of 2 MB and 1 GB have been retrofitted in the kernel with additional hardware support. Huge pages reduce the number of translation entries and further shorten page walks.

Linux has two primary mechanisms to allow applications to leverage huge pages. The HugeTLB subsystem allows a system administrator to allocate a certain number of persistent huge pages which can then be explicitly mapped by userspace applications. HugeTLB requires careful coordination between system administrators and application developers to ensure a proper number of huge pages are available.

Alternatively, Transparent Huge Pages (THP) [204] can be used to allocate huge pages transparently to the application. THPs only support 2MB pages and opportunistically assign huge pages to applications, either in page fault handling or through background page promotion. The OS forms huge pages on top of contiguous physical memory regions that it keeps on free lists. However, userspace memory pages, kernel structures, and other sources can cause a fragmented physical address space with no huge pages available.

Memory Fragmentation. Memory fragmentation can be alleviated by memory compaction, also known as defragmentation. Memory compaction [47] tries to consolidate physical pages by moving them around, freeing up contiguous memory areas for huge page allocations. During movement, the page becomes temporarily unavailable until the page is copied to the new location and the new mapping is established.

Page Migration and TLB shootdowns. Figure 2.1 shows the process of page migration. A translation for a page is cached in the TLBs. However, in contemporary processors TLBs are not cache-coherent. Hence, the OS needs to invalidate all potential *victim* TLBs of cores that might be caching the translation. First, in Step ① the OS running on top of the *initiator* core clears the present bit of the page table entry of the page under migration. Then, the initiator invalidates the entry from the local TLB and initiates the TLB shutdown procedure, in Step ②. Specifically, the OS sends inter-processor interrupts (IPIs) in Step ③ to all remote victim cores. Each core receiving an IPI invokes an interrupt handler that flushes their private TLB, shown in Step ④, and then sends an

acknowledgment in Step ⑤. After the initiator receives all the required acknowledgments, it performs the copy of the page in Step ⑥. Finally, it updates the PTE in Step ⑦.

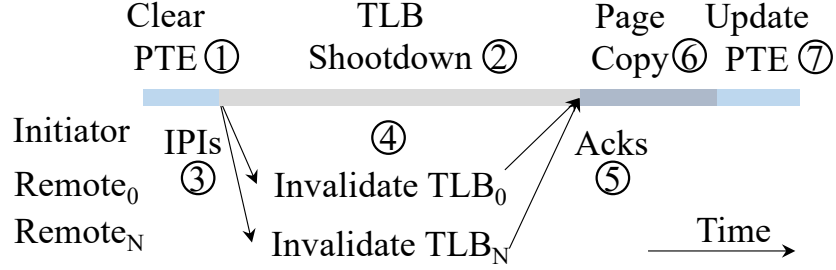


Figure 2.1: Page migration procedure.

TLB shutdowns are very costly as they scale poorly with the number of involved cores and may require several thousands of cycles to complete [12, 13, 20, 26, 115, 115, 177, 211]. Upcoming hardware [10, 18, 88] aims to reduce the TLB invalidation overhead, but a page still becomes unavailable during migration. The same steps are followed by processors that make use of IOMMUs with IOTLBs [9, 90, 159] and NICs with private TLBs [163, 202].

Unmovable Allocations. An impediment to memory compaction is *unmovable allocations* that cannot be migrated. Unmovable allocations exist across different operating systems, called wired allocations in FreeBSD, and non-paged in Windows [146, 179, 225]. While there are many sources of unmovable allocations such as slab, page tables, and other kernel structures, they can be categorized into two types. The first type of unmovable allocations are formed because the kernel opts for faster translation of kernel objects through a simple offset into a linear map [205]. Addresses to these data structures may be stored all over memory. As a result, such data structures cannot be moved.

Other types of unmovable allocations are related to I/O and the IOMMU, including memory for kernel-bypass networking and storage, GPUs, accelerators, and in general operations that mark pages as busy and thus unmovable. Such allocations are impossible to move with software alone. The reason is that the software cannot atomically perform both the translation update and the page copy. Hence, the only way for software to migrate a page without potential spurious writes to the page taking place during migration is to first block access to the page, perform the copy, and then update the translation. However, access to such unmovable pages cannot be blocked as incoming operations from a device need to be serviced. Even if access to the page could be blocked, page migration itself is expensive because it requires a long downtime to (a) perform a TLB shutdown procedure that scales poorly with the number of victim TLBs, and (b) copy the page.

2.2.2 Memory Capacity and TLB Trends

Trends in Meta’s hardware show increases in memory capacity without comparable increases in TLB capacity. This discrepancy puts increased emphasis on contiguity availability to reduce address translation overheads.

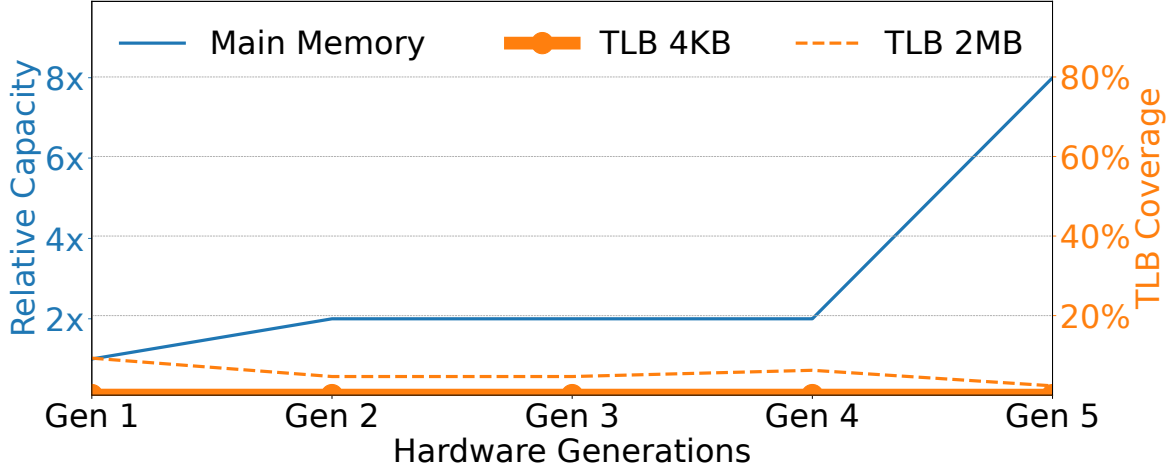


Figure 2.2: Memory and TLB coverage of computing hardware across generations.

Figure 2.2 shows the relative increase of memory capacity and TLB coverage of computing infrastructure across hardware generations in Meta’s data centers. The x-axis shows different hardware generations. The *Gen-1* hardware is near its end of life while *Gen-4* and *Gen-5* are expected to be deployed in the near future. We normalize the results based on the first generation. Memory capacity is bound to increase by almost $8\times$. However, the number of TLB entries and consequently TLB coverage remain stagnant. Specifically, the number of TLB entries has steadily remained in the range of a few thousands in the last few generations. As a result, TLB coverage with contemporary 4 KB pages, and even larger 2 MB pages will be significantly insufficient. 1 GB pages do provide sufficient coverage that is larger than the main memory capacity of *Gen-5* hardware. Reducing translation overheads will require ever larger page sizes, and hence contiguity, in hand with techniques to reduce page walk latency.

2.2.3 Lost Cycles Due to Lack of Contiguity

To quantify the potential of memory contiguity we select a few representative services across Meta’s fleet and measure the impact of contiguity. We perform measurements using: i) only 4 KB pages, ii) 2 MB pages, iii) 2 MB and 1 GB. For 2 MB we use THP and further reserve huge pages through HugeTLB [206] for services that have already been optimized to support HugeTLB. To leverage 1 GB pages we solely rely on HugeTLB. We focus the evaluation on the most widely used machine type which has 64GB memory. At Meta, services are deployed directly on Linux servers within containers. Furthermore, services are optimized heavily to fit within the available memory. Hence, the results provide a lower bound of the potential address translation overhead and gains from larger page sizes—we expect such overheads to be further exacerbated as memory capacity increases.

Figure 2.3 shows page walk cycles as percentages of the total cycles, due to *Data* and *Instructions*, using performance counters of production workloads. For *Data* we show the

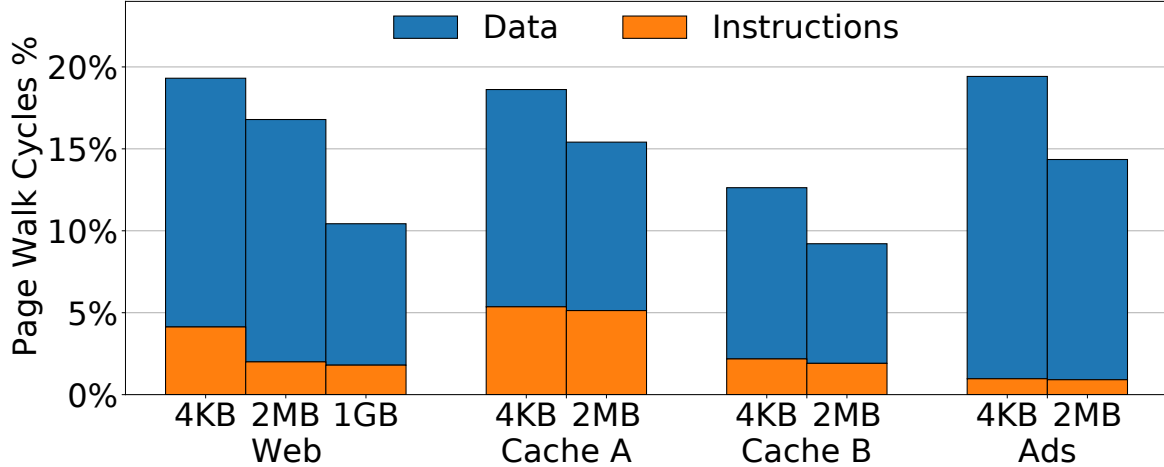


Figure 2.3: Percentage of cycles lost due to page walks.

aggregate of loads and stores. We observe that page walk cycles can account for close to 20% of total cycles in Meta’s data centers. Memory contiguity has the potential to significantly alleviate TLB overheads of both instructions and data. Focusing on Web, one of the largest services within Meta, we see that 2 MB huge pages can halve the number of instruction page walk cycles. Notably, while 2 MB pages offer little improvement for data page walk cycles, 1 GB huge pages have a major impact by reducing such cycles from 14% down to 8%.

2.2.4 Memory Fragmentation

Memory fragmentation prevents the OS from creating large contiguous regions. To quantify fragmentation across Meta’s fleet, we randomly sample tens of thousands of 64GB servers in production independently of workloads and perform a full scan of each server’s physical memory. Figure 2.4 shows the cumulative distribution function (CDF) of memory contiguity as a percentage of free memory at the 2 MB, 4 MB, 32 MB and 1 GB allocation levels. Servers having less than 1 GB of free memory are filtered out. We observe that memory fragmentation is severe across the fleet i.e., 23% of the servers do not have enough contiguity for even a single 2 MB allocation. This number increases to 59% for 32 MB allocations. Dynamically allocating 1 GB pages is practically impossible in production.

Correlation With Uptime. It is commonly assumed that memory fragmentation is correlated with server up-time [155, 225]. In particular, freshly brought-up servers are expected to have an abundance of available memory contiguity that *slowly* deteriorates as workloads execute. Interestingly, our analysis showed there is little to no correlation between memory contiguity availability and server uptime, with the Pearson correlation coefficient between server uptime and the number of free 2 MB pages being only 0.00286. We further looked into servers with only a few hours of uptime and discovered that the correlation still remained weak, with a coefficient of 0.16. Finally, we performed a set of

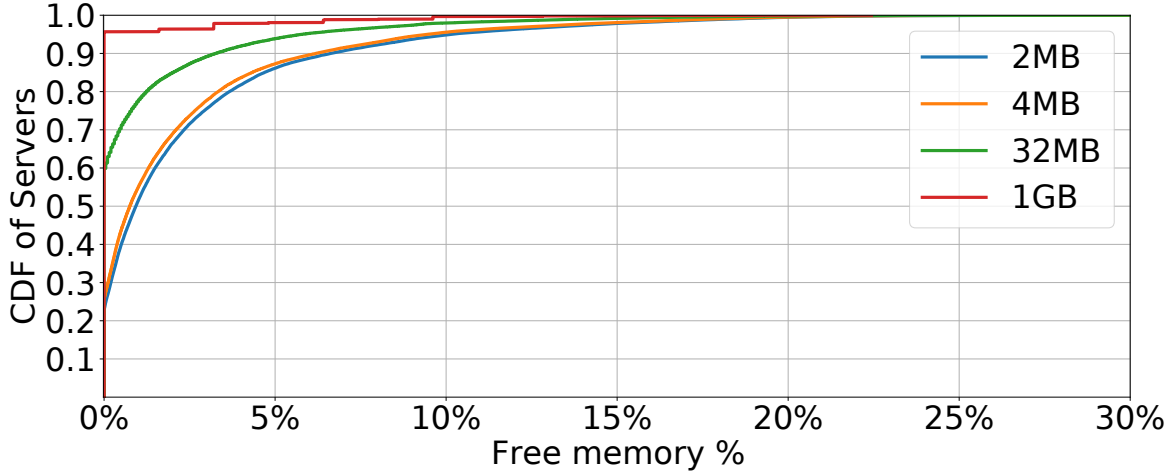


Figure 2.4: Contiguity availability as the percentage of free memory.

experiments that indicated that servers can get highly fragmented within the first hour of running workloads. Given that the mean uptime of Meta’s servers is multiple days or weeks, memory fragmentation affects nearly all servers.

2.2.5 Unmovable Memory Allocations

Despite high amounts of fragmentation, the kernel could succeed in allocating huge pages by compacting allocated pages into fewer contiguous regions. Unmovable allocations inherently limit memory contiguity as they impede memory compaction. To quantify unmovable allocations we follow the same process for studying memory fragmentation. Figure 2.5 shows the percentage of 2 MB unmovable pages relative to the total memory. As we can see, a median server has 34% of its memory occupied by unmovable allocations. As a result, a significant portion of the memory cannot be used for huge pages, and forming any larger contiguous regions (at 4 MB, 32 MB or 1GB) will be even harder. Furthermore, the variance across machines is a major obstacle for using huge pages. At Meta it is desirable to treat the servers interchangeably so that a workload can potentially land on any available server in the fleet. In practice, for some critical services that depend on the existence of sufficient huge pages, automatic server reboots are used to resolve high fragmentation.

Furthermore, we identified that unmovable allocations are *scattered* across the address space. The median ratio of the number of unmovable 4 KB pages over the total number of pages is only 7.6%, but it makes 34% of 2MB pages unmovable, showing that scattering is greatly exacerbating the unmovable memory issue.

To identify the sources of unmovable allocations, we track all allocations marked as unmovable and backtrace their allocation sources. Figure 2.6 shows the breakdown of various sources of unmovable allocations. Networking-related operations are a major source of unmovable allocations, accounting for more than 73%. Such allocations include send and receive buffers maintained by the OS. These buffers carry the data received or to be sent

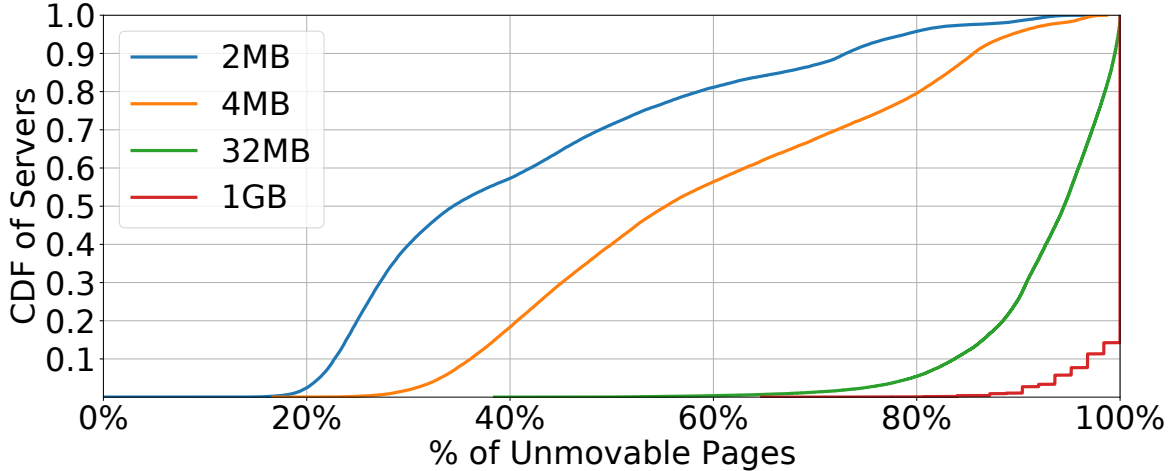


Figure 2.5: Distribution of unmovable pages in contiguous regions of 2MB, 4MB, 32MB, and 1GB.

through the processing of different layers of the networking stack, from the applications that own the sockets down to NICs [80]. In Meta’s environment the number of networking-related pages constantly remains high. More importantly, we expect unmovable networking allocations to become an increasingly bigger problem for two reasons. First, with the adoption of high bandwidth NICs [201] and the increased number of queues based on the number of cores, the number of networking allocations are expected to increase drastically. Second, with kernel bypass and RDMA technologies [59, 73, 144, 166, 241] in data centers, networking pages are pinned to memory and remain unmovable for the lifetime of an application.

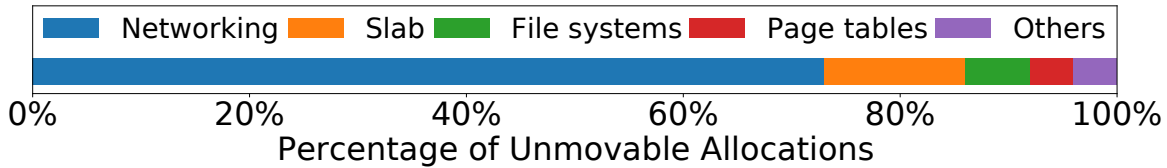


Figure 2.6: Sources of unmovable allocations.

The second source, which accounts for 12%, is related to slab allocations. The slab allocator is the small object allocator in the Linux kernel that packs objects in pages obtained from the page allocator. It is the primary memory source for kernel data structures which cannot be moved. File systems frequently allocate pages as buffers for compression and decompression. Page tables are used to store translation entries from virtual addresses to physical addresses. About 4% of allocations are related to other sources.

In the future, we expect additional sources of unmovable memory driven by the increased deployment of heterogeneous hardware such as GPUs, accelerators, and other

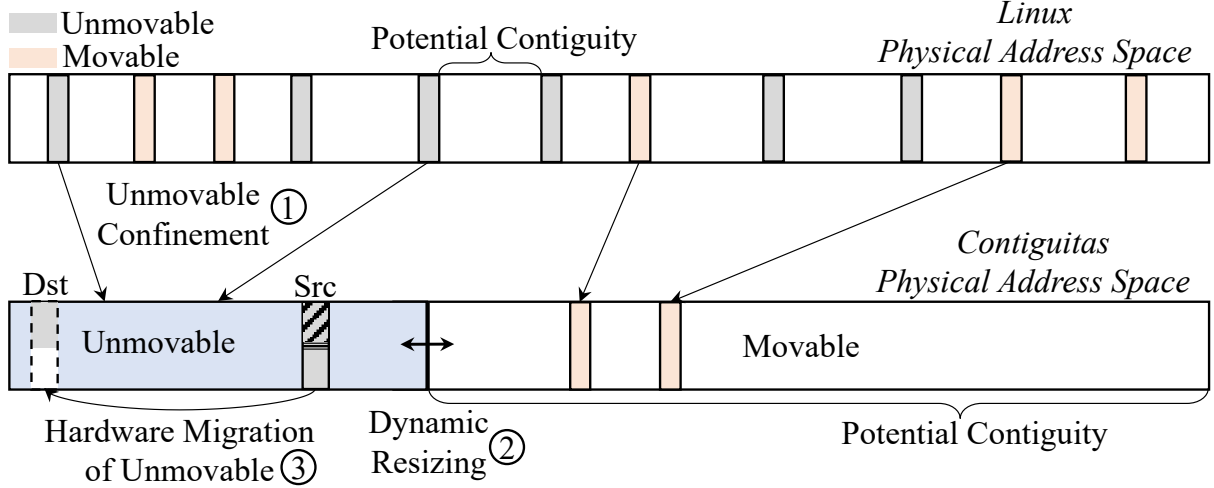


Figure 2.7: Contiguitas design overview.

devices that rely on unmovable allocations. As a result, managing unmovable allocations is critical for efficient memory management.

2.3 Contiguitas Design

The goal of Contiguitas is to provide ample physical memory contiguity by reducing memory fragmentation due to unmovable allocations. To that end, Contiguitas redesigns memory management in the OS to confine unmovable allocations and completely separate them from movable ones. In addition, Contiguitas drastically reduces unmovable pages in data centers. Specifically, Contiguitas introduces a set of hardware extensions in the last-level cache (LLC) that enable the transparent migration of unmovable pages while in use.

2.3.1 Overview

Figure 2.7 provides a high level overview of Contiguitas and how it transforms the physical address space. There are two key design principles guiding Contiguitas. The first one, is to strictly separate unmovable from movable allocations using two dedicated regions and, thus, prevent unmovable allocations from scattering across the address space. The result is shown in Step ①. Next, Contiguitas introduces a dynamic resizing mechanism in Step ②, that monitors the demand for allocations in each region and grows or shrinks the corresponding region accordingly. Contiguitas further reduces internal fragmentation of the unmovable region by placing certain types of allocations away from the region’s border.

The second design principle of Contiguitas, is to drastically reduce the amount of unmovable allocations by turning them into movable ones. Our study at Meta’s data centers in Section 2.2.5 revealed that a significant portion of unmovable allocations are

impossible to move as access to the page cannot be blocked for a software migration to take place. Hence, a software solution is inadequate. To this end, Contiguitas introduces a set of hardware extensions in step ③ that enable the transparent migration of unmovable allocations. Naturally, these hardware extensions can further be leveraged by movable pages.

2.3.2 Confining Unmovable Allocations

Contiguitas separates the physical address space into two continuous regions, one for movable and one for unmovable memory. Allocations are confined in their respective region. Contiguitas categorizes the physical pages based on their addresses and keeps them on distinct free lists for each region. Memory in each region can only be allocated from pages in the free lists belonging to that region. When a page is freed, it is returned back to its respective list. This approach simplifies the critical path of allocations as the OS can quickly pick a free page while avoiding mixing different types of allocations.

Confinement is sufficient for allocations that are deemed unmovable during their whole lifetime. However, some allocations might first be allocated as movable but later become unmovable. For example, due to operations such as memory pinning for zero-copy network send operations. In such scenarios, Contiguitas first migrates them to the unmovable region and then marks them as unmovable. This approach avoids the dynamic pollution of the movable region and subsequent compaction failures.

The major challenge in designing confinement is the sizing of the unmovable region. If it is too big, unused memory in the unmovable region is wasted while there is limited movable memory for the applications, causing frequent reclaims, swapping, or even allocation failures. On the other hand, if the unmovable region is too small, it may fail unmovable allocations. Therefore, Contiguitas needs to dynamically balance the sizes of the movable and unmovable regions.

Dynamic Region Resizing. As the need for movable or unmovable memory may change across applications and their phases, static sizing would be ineffective. To this end, Contiguitas dynamically resizes the two regions to keep the unmovable region small while not negatively affecting application performance. Contiguitas sets up the physical memory layout during system boot-up, and gives the unmovable region a configurable initial size based on the memory capacity of the system and the behavior of workloads in our fleet. For our 64GB servers, we configure this initial size to 4GB. The boundary of the two regions is then tracked in the OS to support resizing. To expand the unmovable region, Contiguitas adjusts the boundary, moves away allocated pages, and takes over space from the movable region, adding pages to the unmovable free list.

Dynamic resizing of the unmovable region introduces three major design challenges. The first challenge is to move resizing operations off the critical path of memory allocation. The second challenge is to choose the placement policy of unmovable allocations within the unmovable region and avoid internal fragmentation. Allocations exhibit various lifetimes and as a result, placing an unmovable allocation with a long lifetime close to the boundary, might unnecessarily block the shrinking of the region even if free space is abundant within the region. The last challenge is to choose the proper size of the unmovable region, which

Algorithm 1: Region resizing algorithm.

```
1 function RESIZE( $Pressure_{unmov}$ ,  $Threshold_{unmov}$ ,  $Pressure_{mov}$ ,  $Threshold_{mov}$ ,  
    $c_{ue}$ ,  $c_{us}$ ,  $c_{me}$ ,  $c_{ms}$ )  
2   if  $Pressure_{unmov} \geq Threshold_{unmov}$  and  $Pressure_{mov} < Threshold_{mov}$  then  
3     // Expand unmovable upon high pressure  
4      $F \leftarrow \frac{Pressure_{unmov}}{Threshold_{unmov}} \cdot c_{ue} + \frac{Threshold_{mov}}{\max(Pressure_{mov}, 1)} \cdot c_{me}$   
5      $U \leftarrow (1 + F) \cdot Mem_{unmov}$   
6   else  
7     // Shrink for all other cases  
8      $F \leftarrow \frac{Pressure_{mov}}{Threshold_{mov}} \cdot c_{ms} + \frac{Threshold_{unmov}}{\max(Pressure_{unmov}, 1)} \cdot c_{us}$   
9      $U \leftarrow (1 - F) \cdot Mem_{unmov}$   
10  return  $U$ ;
```

is workload dependent.

Contiguitas performs resizing off the critical path of memory allocation to avoid latency overheads. This is accomplished by monitoring the amount of free memory when periodic memory reclaim is triggered by the kernel. Contiguitas extends reclaim to wake up a kernel thread to perform resizing when the free memory in either region falls below a low-watermark threshold.

In addition, Contiguitas introduces a bias to prefer physical pages further away from the region border. Some unmovable allocations that are inherently long lived, e.g., kernel code pages, are safely placed by Contiguitas early on, at the end of the unmovable region that is farthest from the movable region. On the other hand, pages that are initially in the movable region and later on migrated to the unmovable region often exhibit short lifetimes. In general, Contiguitas prefers allocating pages away from the region border as long as sufficient free space is available. This approach increases the chance that shrinking will be successful. Later on, in Section 2.3.3 we will further introduce the hardware extensions of Contiguitas that enables the migration of unmovable pages.

Finally, to aide in deciding when and how much to resize, Contiguitas introduces the concept of per-region memory pressure, which builds on top of the kernel's pressure stall information (PSI) used for reclaim [216]. PSI tracks the percentage of time wasted due to lack of free memory (e.g. due to page faults of recently resident memory, or direct reclaim). We extend PSI to track time wasted due to lack of free memory in both the movable and unmovable region separately.

The resizing algorithm is depicted in Algorithm 1. Given per-region pressure, $Pressure_{unmov|mov}$, configurable thresholds,

$Threshold_{unmov|mov}$, and coefficients that fine-tune the expansion and shrinkage of the region, c_{ue} , c_{me} , c_{us} , and c_{ms} , the algorithm resizes the unmovable region. Depending on pressure the kernel expands or shrinks the unmovable region. Contiguitas sets parameters for dynamically resizing empirically by observing the patterns for movable and unmovable allocations of the workloads, and tunes the parameters iteratively. Contiguitas uses global parameters that work well for a diverse set of workloads, and we leave automated

parameter space search as future work.

2.3.3 Architectural Support for Transparent Page Mobility

The goal of the hardware extensions of Contiguitas, Contiguitas-HW, is to enable the migration of the subset of unmovable pages that must remain accessible even during migration. Such pages represent a major chunk of allocations in data centers that is bound to increase in the near future (Section 2.2.5). Hardware support is required because it is impossible for software to move such pages as it cannot atomically perform both the translation update and the page copy operation. Hence, software has to block access to the page for the duration of page migration in order to avoid spurious writes to it. Even if access to the page could be blocked, software page migration induces a long downtime due to (a) TLB shutdowns that scale poorly with the number of victim TLBs, and (b) the page copy.

To this end, Contiguitas-HW enables transparent page migration while the page remains in use. Such migrations can substantially reduce the size of the unmovable region and lead to more efficient defragmentation and memory management as the vast majority of pages can be moved on demand. While Contiguitas-HW is motivated by unmovable allocations, its design is suitable for both movable and unmovable allocations.

Platform Overview. Figure 2.8(a) shows a high-level overview of the hardware platform of Contiguitas-HW. The hardware extensions of Contiguitas-HW are located in the LLC. Contiguitas-HW targets environments that are representative of current and upcoming hardware platforms such as those based on CXL [46]. Specifically, Contiguitas-HW targets a multi-core processor with a cache-coherent interconnect. The hardware platform further includes an IOMMU [9, 90] with local TLBs. The IOMMU performs similar duties to the MMU on the CPU side, such as page walks and caching of translations in the TLB for devices. The NIC includes a TLB and is cache-coherent with the LLC. The core performs IOTLB shutdowns by putting invalidation requests onto a queue in memory [90]. Device TLBs cache translations from the IOMMU, and the core can synchronize them with the IOMMU by submitting invalidation requests through the aforementioned queue.

Hardware Operations. At a high level, Contiguitas-HW aliases a physical page under migration with a destination page and redirects appropriate traffic to the destination page based on the progress of the migration.

Specifically, a page migration is initiated in Step ① by the OS that provides the source and destination physical page numbers (PPNs) to the Contiguitas-HW. Contiguitas-HW stores them in a metadata table shown in Figure 2.8(b). In addition, Contiguitas-HW stores a *Ptr* field that points to the next line to be copied [28, 169, 188]. It effectively tracks the number of cache lines that have been copied.

Before a cache line can be copied, Contiguitas-HW issues *BusRdX* events for both the source and the destination cache lines, shown in Step ②. This step guarantees that the most recent version of the source cache line is located in the LLC and that both source and destination cache lines are invalidated in higher level caches. Next, Contiguitas-HW copies the line from the source to the destination, as shown in Step ③. Then, it increments *Ptr* by one. This process continues until the page is completely copied.

optimization is useful because at any point a core may issue requests based on the source mapping found in the TLB while Contiguitas-HW may respond based on the data in the destination mapping.

While it is possible to allow the caching of the line in the private caches, this approach would introduce two implications: First, it would require Contiguitas-HW to keep track of such lines to maintain coherence. Second, writes performed to the source page would have to be propagated to the destination as well. Noncacheable accesses resolve this issue as all traffic is redirected to the LLC.

Noncacheable accesses are already supported by contemporary processors [96]. In existing processors, page table entries maintain a page-level cache disable (PCD) bit that is also installed in the TLB. Requests for a page with the PCD bit set in the TLB are performed as noncacheable accesses. However, in current processors such accesses bypass the complete cache hierarchy, including the LLC. Contiguitas-HW piggybacks on existing hardware support for noncacheable accesses, but instead selectively activates noncacheable accesses while allowing caching in the LLC.

At a high level, one possible approach would be for Contiguitas-HW to set the non-cacheable bit when a line is returned from the LLC to the private caches, notifying them to not cache the line. However, since noncacheable information is available a priori, at translation time, noncacheable requests issued by the core may follow a different and simpler path when traversing the cache hierarchy.

To this end, Contiguitas can integrate with existing noncacheable mechanisms as follows. When Contiguitas-HW issues a *BusRdX* it also notifies the cache agents in the private caches to install an entry in their miss status holding registers (MSHRs) corresponding to the source PPN, marking it as noncacheable. When the migration is complete Contiguitas-HW notifies the private caches to clear the entry. Finally, there is one special case that Contiguitas-HW needs to handle—a core that has not received an invalidation issues a request for the source page for the first time. In this case, Contiguitas-HW nacks the request and notifies the private caches as before. Then the cache agent will retry the cache access as noncacheable. This mechanism can further be implemented on top of upcoming hardware extensions [88] that will support efficient and extensible IPIs in hardware.

Cacheable Accesses. Noncacheable accesses resolve the challenge of maintaining coherence for a line that can be accessed by different mappings. However, with careful examination of the migration process, we identify that private caching can stay enabled as long as only a single mapping is used to cache a line at any given time. The benefit of this approach is twofold. First, lines under migration can reside in private caches. Second, it eliminates the need to notify the private cache agents to handle migration pages as noncacheable.

To achieve this goal, Contiguitas-HW performs the following steps. First, when the OS notifies the Contiguitas-HW to start migrating a page, the hardware enables traffic redirection but does not start the copy process. Instead, the OS immediately modifies the page table entry to point to the destination page and starts invalidating each TLB. During this process, a page may be accessed with either the source or the destination mapping. If a request hits in the private caches, it is serviced without further action as in a regular cache hit. Otherwise, on a miss the Contiguitas-HW checks whether a line is currently

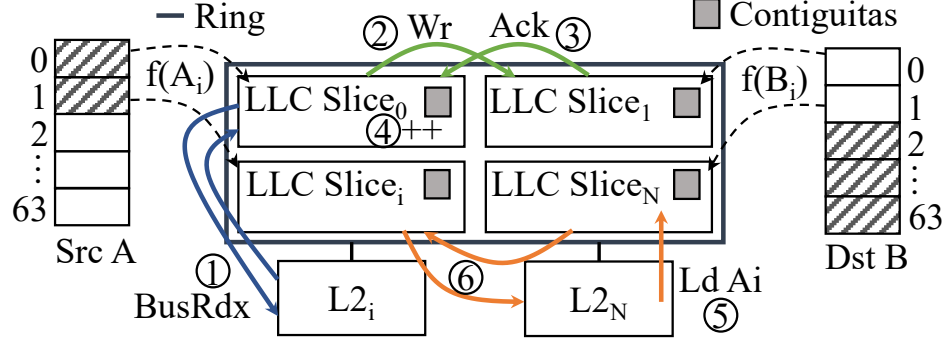


Figure 2.9: Contiguitas operations in a sliced last-level cache architecture.

stored in private caches with the opposite mapping of the request i.e., if a request is for the source mapping and the line is stored with the destination mapping, and vice versa. If the condition is true, the request invalidates any cached copy. Otherwise, the request is serviced regularly. This invariant allows the caching of lines under migration as only the source or the destination mapping is active in the private caches.

When the TLB invalidations are complete, the OS notifies the Contiguitas-HW to start the copy. At this point only the destination mapping is active at any given TLB. During the copy process, a request for the destination mapping might be in a modified state. In that case, the copying of that line is skipped as the destination already contains the most up-to-date data. Other operations are handled regularly. By the end of the migration the Contiguitas-HW notifies the OS as before.

Distributed Last-level Cache Slices. So far we have discussed the operations of Contiguitas-HW in the context of a monolithic last-level cache (LLC). In practice, the LLC usually consists of multiple slices that communicate over an interconnect such as a ring bus [96]. Figure 2.9 shows a high level overview of a sliced cache architecture. In this environment, a physical address is passed through a hash function f that spreads the cache lines of the page across different slices. As a result, a given source page A and destination page B, $f(A_i)$ and $f(B_i)$, may map each of their source and destination cache lines to different slices.

In this environment, Contiguitas-HW and its metadata table are replicated across each slice. In addition, Contiguitas-HW adds a copy of the slice selection hash function. This is needed because the hash function is usually located at the private L2 caches. The added hardware is cheap because such hash functions are a simple combination of logic gates e.g., XOR [229]. To use the *Src PPN* to initiate the copy of the lines, it first computes the slice selection function, $f(A_i)$ and checks if the address corresponds to the local slice.

If the line belongs to this slice, then Contiguitas-HW issues a *BusRdX* event as before, in Step ①. When the line is invalidated in the private caches and the most recent version of the line is in the LLC slice, Contiguitas-HW uses the *Dst PPN*, $f(B_i)$ to identify the “home” slice of the destination line. If the home is the current slice then the copy is performed locally as before. Otherwise, Contiguitas-HW sends a Write request to the destination slice, shown in Step ②. After the write is complete, the destination slice responds in

Step ③ with an acknowledgement. At this point, in Step ④ Contiguitas increments Ptr . If the line does not belong to this slice, Contiguitas-HW increments Ptr until it reaches a line that belongs to the local slice. This is correct because each local slice is only responsible for the lines within the slice.

Meanwhile, a request for the source page A_i may arrive at a slice, shown in Step ⑤. The slice responds in the following manner. First, it compares the address to Ptr to identify the migration status of the line as before. If the line has been migrated then Contiguitas-HW uses $f(B_i)$ to identify the destination slice. If the destination slice is the current one, then the operation continues locally as before. Otherwise, Contiguitas-HW does not need to perform the lookup in this slice. Instead, in Step ⑥ it forwards the request to the destination slice that then responds to the request. Overall, this process is similar to handling the case that a request arrives at an LLC slice but the line is dirty in a private cache.

The final piece of this process is how slices coordinate on when to perform their local copy procedures for a given page. With the above protocol in place and the natural parallelism provided by a distributed cache architecture, each slice could operate independently. This approach would significantly improve the performance of the copy operation. However, it would also add additional pressure to the interconnect during the copy operation. Instead, given the target operation of Contiguitas-HW of moving unmovable allocations, we opt for a simpler and less aggressive design. Specifically, after each slice completes its portion of the copies, it then sends a notification to the next slice to start. In the future, if an even lower latency page copy might be beneficial, leveraging slice parallelism might be more attractive.

Variable Buffer Sizes. Depending on the application characteristics it might be beneficial to support variable mapping sizes for device TLBs. Contiguitas-HW naturally supports this use case. When a mapping is composed of smaller independent translations, Contiguitas-HW can independently move each mapping one at a time. Alternatively, the metadata table of Contiguitas can be extended to maintain a *Size* field that defines the range of the translation. Depending on how the OS opts to map the allocation on the CPU side, the OS can perform TLB updates at a finer granularity than device TLB ones. Overall, Contiguitas-HW enables a large optimization space depending on the application characteristics.

Interface Contiguitas-HW supports two commands to interact with the OS. **Migrate** (PPN_Src , PPN_Dst , $Flag$) sets up a migration mapping by providing the source and destination physical page numbers. Based on the flag argument the mapping is simply installed or a migration is initiated based on the cacheable model of the microarchitecture. **Clear**(PPN_Src) clears the metadata table entry, ending a page migration.

Contiguitas-HW accepts commands from the OS via a work queue similarly to Intel’s DSA [92]. The OS prepares a work descriptor in memory containing commands and parameters to Contiguitas-HW, and submits the descriptor to the work queue using the existing **ENQCMD** instruction [53]. The work descriptor contains a completion address that Contiguitas-HW writes to indicate to the OS that a work is completed.

2.4 Evaluation Methodology

To evaluate Contiguitas, we follow a two-pronged approach. We evaluate the OS components of Contiguitas in Meta’s production environment and further perform full-system simulations for the hardware components.

Production Environment. Unless otherwise noted, the hardware used for the production experiments is Intel Cooper Lake single-socket 64GB servers running Linux 5.12. We implement Contiguitas on top of Linux 5.12. In the rest of this section, we refer to the default kernel as *Linux*, our kernel implementation on top of Linux as *Contiguitas*, and the combination of our kernel implementation and hardware extensions as *Contiguitas-HW*. Since it is hard to run our production workloads on a simulator, we evaluate Contiguitas-HW with a page migration microbenchmark on top of NGINX [172], and memcached [60].

Production Workloads With Live Traffic. Our production evaluation leverages three major workloads at Meta, *Web*, *Cache A*, and *Cache B*. *Web* is a web server and one of the largest applications in deployment across Meta’s fleet. *Cache A* and *Cache B* are the two largest in-memory caching services that handle traffic for most of the internal services at Meta. *Cache B* is a forked version of memcached. We leverage production-grade load-testing A/B infrastructure that has been developed internally at Meta to evaluate hardware selection and software optimizations. During our experiments, all workloads receive *real production traffic*.

Full-system Simulation. To evaluate the hardware components of Contiguitas we use full-system cycle-level simulations. We integrate QEMU [29] with SST [176] and DRAM-Sim3 [124]. We model a server architecture with 8 cores, 64GB of main memory. We run Linux 5.12 as the baseline and compare it with Contiguitas-HW. The architecture parameters are shown in Table 2.1. Each out-of-order core has private L1 and L2 caches, and a slice of the shared L3 cache. In addition, each core has private L1 and L2 TLBs and fully associative (FA) page walk caches. For Contiguitas’s metadata table we use a fully associative structure with 16 entries per slice. To model TLB invalidations we intercept *INVLPG* instructions [96] issued by the OS and perform a traversal of the TLB hierarchy and page walk caches. Through measurements on a real processor, we find that the cost of the *INVLPG* instruction is substantially higher than the TLB round-trip and close to 250 cycles. We believe that it is due to the interactions of the instruction with the core pipeline. We model this behavior by adding a fixed penalty of 250 cycles, accounting for the cost of a full pipeline flush when an *INVLPG* instruction is issued.

Table 2.1: Contiguitas architectural parameters.

Full-System Simulation Parameters	
Multicore Chip	8 4-issue OoO cores, 200-entry ROB, 2GHz
L1 Cache	32KB, 8-way, 2 cycles round trip (RT), 64B line
L1 TLB	64 entries, 4-way, 2 cycles RT
L2 TLB	1536 entries, 16-way, 12 cycles RT
Page Walk Cache	3 levels, 32 entries per level, FA, 2 cycles
L2 cache	256KB, 8-way, 14 cycles RT
L3 cache	2MB slice, 16-way, 40 cycles RT
Contiguitas-HW	16 entries, FA, 1 cycle
Main memory	64GB, DDR4 3200, 16 Banks, 1GHz
OS	Linux 5.12

2.5 Evaluation

Our evaluation attempts to answer the following questions:

1. Does Contiguitas improve application’s end-to-end performance?
2. What is Contiguitas’s impact on unmovable allocations?
3. What is Contiguitas’s impact on memory contiguity?
4. Does page migration in Contiguitas-HW scale with the number of cores?
5. How do Contiguitas-HW page migrations affect page availability and application performance?
6. What are the hardware resource requirements of Contiguitas-HW?

2.5.1 End-to-end Performance Impact

We evaluate the performance of three major workloads at Meta, *Web*, *Cache A*, and *Cache B*. The performance metric is requests per second under certain latency SLAs based on the characteristics of each workload. In all cases, THP is set to “always”. Additionally, Web attempts to allocate 2MB and 1GB huge pages directly.

We consider two setups, *Full Fragmentation* and *Partial Fragmentation*. *Full Fragmentation* represents the case where a workload lands on a server whose memory is already fully fragmented. As we showed in Section 2.2.4, 23% of production servers are fully fragmented. To represent this scenario, we run a fragmentation process to fragment the server fully before the workload is deployed.

Partial Fragmentation represents the case where a workload lands on a partially fragmented server that is representative of the majority of servers at Meta. As each workload fragments the address space at different levels, we first run the same workload to fragment the server and then restart the workload before our experiment. This behavior is common in production due to frequent code deployments.

Figure 2.10 shows the results. Contiguitas produces the same result under *Full Frag-*

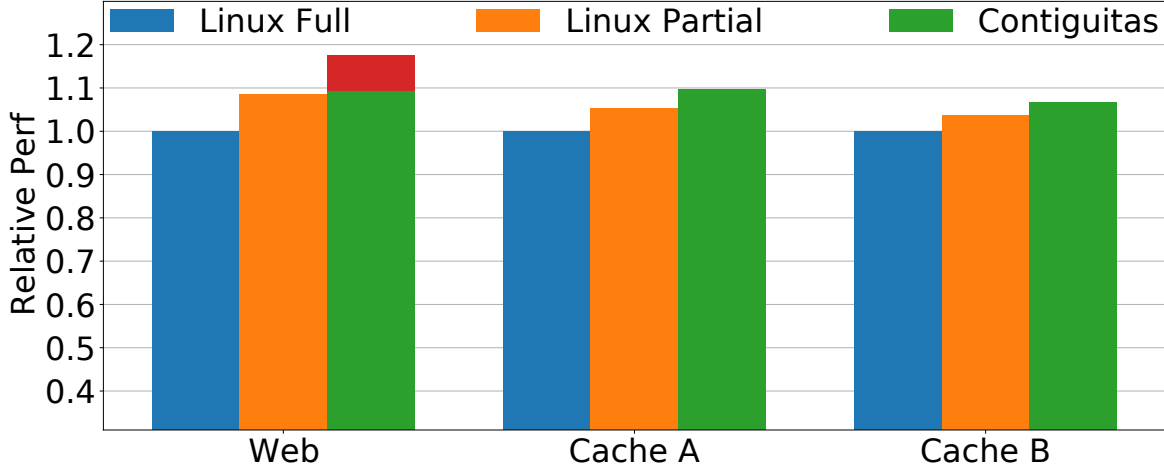


Figure 2.10: End-to-end performance over Meta’s production workloads. The red bar of Contiguitas for Web shows the performance gains from 1GB pages.

mentation and *Partial Fragmentation* and hence we show one bar. In "Linux Partial" and "Linux Full", dynamically allocating 1GB HugeTLB pages always fails due the lack of contiguity. For Web, we show a stacked bar for Contiguitas to highlight the gain with 1 GB pages. Overall, Contiguitas delivers substantial performance improvements between 2% and 18% across major workloads at Meta.

Looking at Web, compared to Linux in the fully fragmented case, Contiguitas achieves 18% higher performance. Compared to Linux in the partial fragmented case, Contiguitas delivers a performance gain of 9%. Notably, looking at the breakdown of Contiguitas we see that substantial gains are due to Contiguitas being able to *dynamically* allocate 1 GB HugeTLB pages. Diving deeper, we find that Linux with partial fragmentation was able to allocate 14 GB worth of 2MB huge pages compared to 20 GB allocated by Contiguitas. For 1 GB huge pages, Contiguitas allocated 4GB in total whereas Linux allocated none. Note that, in Contiguitas, 1 GB pages provided a major performance win of 7.5%, which highlights the importance of bigger contiguity beyond 2MB. Finally, we compared Contiguitas and Linux when both were forced to use only 4KB pages. While the results are not shown in the figure, they had identical performance, showing that Contiguitas introduces no measurable overhead even if it finds no opportunities to allocate huge pages in the worst case.

2.5.2 Unmovable Allocations and Memory Contiguity.

Unmovable allocations. To assess the impact of Contiguitas on unmovable allocations, we compare the percentage of memory that becomes unmovable for different workloads at Meta. We further include Meta’s continuous-integration (CI) workload that executes various build and test jobs. For this experiment, we start our workloads on freshly booted servers and then profile the servers once every 15 minutes.

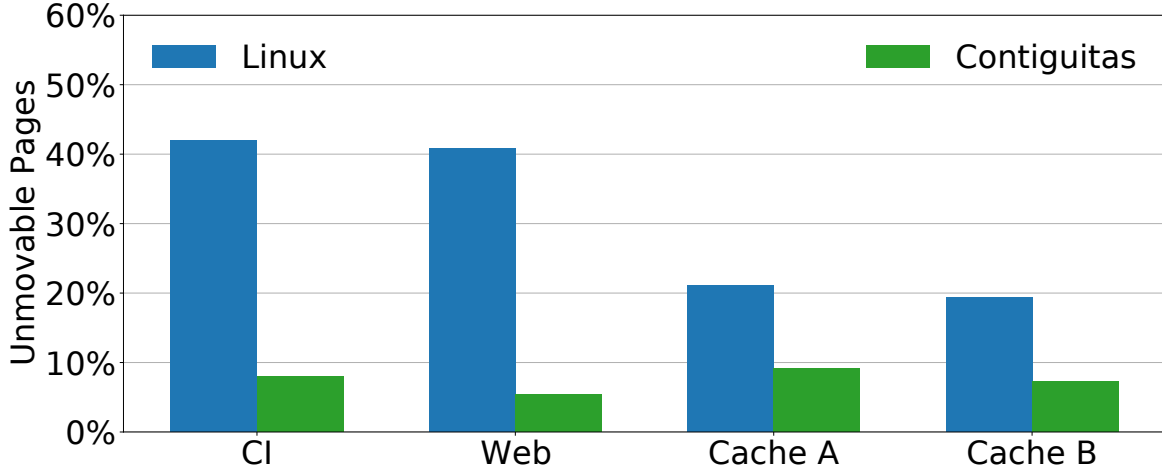


Figure 2.11: Unmovable 2 MB pages for Meta’s production workloads.

Figure 2.11 shows the percentages of 2 MB unmovable pages. We observe that for all workloads we consider, unmovable memory increases drastically within the first hour and then plateaus. Therefore, we show the percentages at their stable state. As we discussed in Section 2.2.4, given the long uptime of servers, fragmentation is ubiquitous across the fleet. Under Linux, the unmovable memory is between 19% to 42%, with an average of 31%. In contrast, Contiguitas has the highest value at 9%, with an average of 7%. This demonstrates that Contiguitas’s confinement strategy successfully restricts the spread of unmovable 2MB pages and keeps the unmovable region small.

To understand the internals of the unmovable region, we characterize its fragmentation. Specifically, we scan the unmovable region and count the number of unmovable pages within each 2 MB block. Note that, in the unmovable region a page is either unmovable or free. We find that 22% of the pages in a typical 2 MB block in the unmovable region are free. This internal fragmentation of the unmovable region further motivates the need for Contiguitas-HW, especially as the number of unmovable allocations increases (Section 2.2.5).

Potential memory contiguity. To quantify the impact of Contiguitas on memory contiguity, we compare each workload’s steady state under Linux and Contiguitas. Specifically, we quantify the contiguous regions that can be formed if we *hypothetically* run a perfect software compaction in order to service allocation requests of 2 MB, 32 MB, and 1 GB. Figure 2.12 shows the results. With Linux we see that some 2 MB allocation are possible given that less than half the memory is composed of unmovable 2MB pages as we discussed above. However, Linux struggles as we search for larger contiguous regions, and fails to find even a single 1 GB page. On the other hand, Contiguitas, by design, isolates the unmovable region and hence the whole movable region can potentially be used after compaction for large contiguous allocations, even 1 GB pages as we showed in Section 2.5.1.

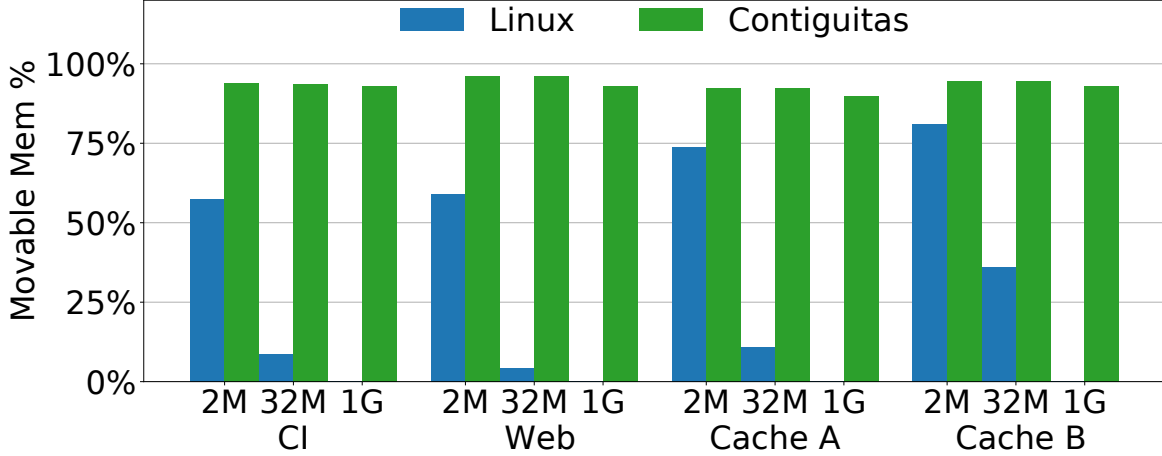


Figure 2.12: Potential memory contiguity as a percentage of total memory.

2.5.3 Contiguitas-HW Characterization.

Scalability. To evaluate the scalability of Contiguitas-HW with respect to the number of cores we use full system simulations. Specifically, we develop a micro-benchmark that triggers page migrations in software and Contiguitas-HW. We simulate all the operations of Contiguitas-HW and instrument Linux to track the page migration procedure, including TLB shootdowns and page copy operations. We then measure the number of cycles a page is unavailable during migration from the perspective of a memory operation to that page. Figure 2.13 shows the results. The measurement with one core means that we perform a TLB shootdown on one remote core. *Linux-Sim* shows the result of a simulated system, which is compared to the result of a real system, denoted *Linux-Real*. We see that the two systems match well (-6% to +10% difference), which confirms the accuracy of the simulation.

From the results we see that as the number of cores/TLBs involved in the TLB shoot-down increases, the amount of time during which access to the page is blocked increases linearly. The cost of the page copy remains the same, at approximately 1,300 cycles. Finally, we plot the result from *Contiguitas-HW*. By design, the cost of Contiguitas-HW is constant and equal to a local TLB invalidation, because its TLB shootdowns do not require inter-processor interrupts and each TLB can be locally invalidated without coordination or synchronous acknowledgements. The cost of a 4 KB page migration in Contiguitas-HW is close to 2 μ s in our microbenchmark. Although the absolute migration time will vary based on the cache behavior of the page, access to the page is never blocked during a migrations.

Performance. We evaluate with full system simulations the performance implications of Contiguitas-HW on two open source applications NGINX and Memcached. We select these workloads to cover both ends of the spectrum. NGINX is insensitive to huge pages and it makes use of networking buffers. In this scenario, our goal is to move unmovable pages that will benefit other applications without interfering with the application that makes use

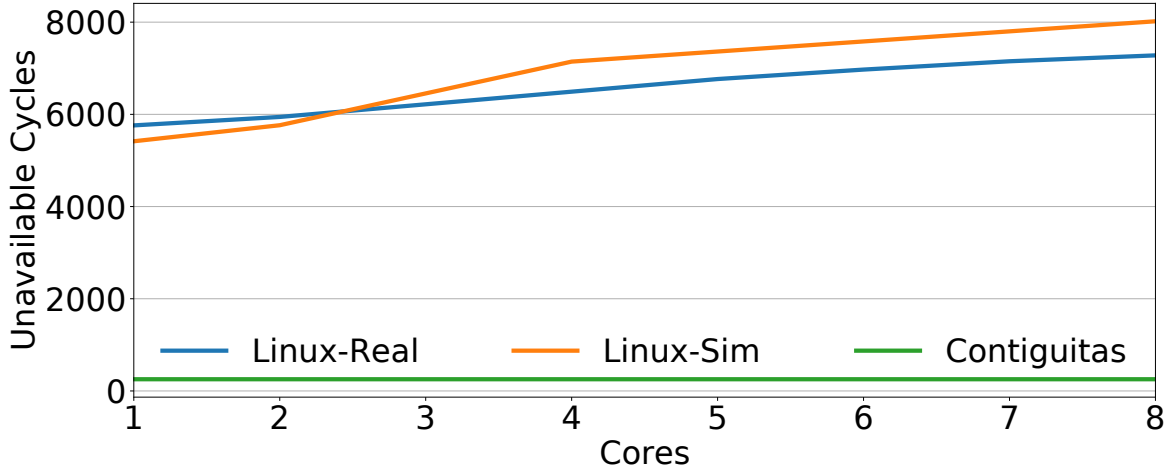


Figure 2.13: Page unavailable cycles during migration.

of unmovable allocations. Memcached is a close proxy to our Cache B workload, which is a fork of memcached; it is sensitive to huge page availability and also relies on networking buffers.

In both cases we consider an aggressively disadvantageous scenario where Contiguitas-HW moves unmovable networking buffers only of the application while the application serves requests at peak throughput without any available slack. We consider two rates: (i) *Regular* is the expected rate of unmovable page movement set at 100/second based on the ratio of unmovable and movable pages, (ii) *Very High* which is the rate if we were to move unmovable pages at the highest rate we observed our production systems to move movable pages, set at 1000/second. We evaluate both workloads without huge pages in order to understand any potential negative performance impact by unmovable page movement without any performance gains from huge pages.

At *Regular* rate Contiguitas-HW with noncacheable and cacheable accesses do not have an impact on application performance. Even at *Very High* rate, which would be unwarranted for a real environment, Contiguitas-HW with noncacheable accesses only shows a marginal overhead of 0.2% for NGINX and 0.3% for memcached. The Contiguitas-HW with cacheable accesses does not have an impact on application performance even at this rate. This is because Contiguitas-HW is able to very efficiently redirect traffic and perform copy operations in the background.

When combined with the benefits of contiguity and 2MB huge pages, memcached performance improves by 7%. Overall, Contiguitas-HW does not negatively impact applications that do not benefit from contiguity while improving contiguity for those that do. In addition, Contiguitas-HW successfully achieves its goal to drastically reduce the unmovable region. Specifically, it enables the migration of the majority of the unmovable pages and further enables the defragmentation of the unmovable region that can potentially unblock 22% of the region’s memory (Section 2.5.2).

Sizing and Hardware Requirements. To size the metadata table of Contiguitas-HW we

consider the following parameters. First, Contiguitas-HW enables local TLB invalidations that can occur whenever the kernel naturally executes on top of a core. Hence, we need to take into account the amount of time a mapping needs to be maintained. To this end we consider the rate of system calls and context switches on a core that is observed to be 40K-100K per second in production. This provides a window of at least 25us for an invalidation to take place. Accounting conservatively for 5us for a 4KB page copying to take place this provides us with 30us per migration. As a result, a single entry already provides a very high theoretical number of migrations/second.

To facilitate concurrent migrations, we size the metadata table to store 16 entries as a fully associative structure. Requests across slices are buffered in existing slice queues that handle regular cache requests. Contiguitas-HW augments messages to pass additional information e.g., noncacheable. However, it does not require additional buses as requests and responses are flowing in existing paths between L3 slices and the L2 caches. Our Cacti [22] analysis for a 22nm technology node shows that the area for Contiguitas-HW per slice is $0.0038mm^2$, the energy per access is $0.0017nj$, and the leakage power is $0.64mW$. We conservatively account for 2-cycle access time to the metadata table. Compared to the size of a core, the cost of Contiguitas-HW is only 0.014%. Overall, the hardware cost of Contiguitas-HW is negligible.

2.6 Discussion

As server memory capacity increases drastically, resolving excessive memory management overheads will require a fundamental rebuild of the operating system and hardware layers. Contiguitas tames unmovable allocations and introduces a reliable source of physical memory contiguity that can enable a plethora of prior [65, 68, 74, 75, 85, 117, 136, 137, 143, 155, 156, 160, 162, 188, 193, 194, 210, 225, 228] and future works that aim to tackle the address translation overhead. In the near term, we expect that 2 MB huge pages can become a primary choice for memory intensive applications and curb the address translation overhead. Finally, existing 1 GB huge pages can provide major performance benefits for certain applications when carefully deployed, however we believe that additional support and research is needed as the substantial increase in page size can exacerbate the severity of data movement overheads and memory bloating.

2.7 Related Work

TLB efficiency has been a major target of prior work [24, 34, 48, 107, 107, 138, 143, 148, 157, 160, 161, 186, 191, 210, 225]. Works have focused on coalesced [161], range-based [107], and part of memory TLB designs [140, 180]. Contiguitas can boost their efficiency by providing ample contiguity.

Huge page support has received significant attention in the past [67, 85, 117, 136, 137, 155, 156, 170, 199, 224, 225]. Other efforts have focused on upstreaming 1GB THP support [224]. As we showed in Section 2.2.2, 1GB huge pages can be effective in reducing

the address translation cost. Work on capabilities [218] can reduce address translation overheads; the approach of Contiguitas could be useful in such environments in order to efficiently manage DMA and other sources of unmovable pages. More recent proposals have also focused on improving the efficiency of userspace allocators to better utilize huge pages [85, 136, 137]. Contiguitas can further boost the efficiency of userspace allocators as they rely on the OS to provide contiguity.

Another body of work has focused on improving TLB shutdown operations [12, 13, 20, 26, 115, 115, 177, 211]. For example, UNITD [177] proposes TLBs to be included in the regular cache coherence protocol in order to reduce their overhead. LATR [115] proposes asynchronous TLB invalidation’s for operations that are amenable to lazy TLB shutdowns but this mechanism is not applicable to unmovable pages because such mappings cannot afford to become unavailable. Overall, the primary focus of these works is to perform fast TLB shutdown operations so their frequency can be increased [20]. Instead, the goal of Contiguitas is to enable transparent migration of unmovable pages while in use. As a result, Contiguitas arrives at a vastly different design point that requires minimal hardware changes in the last-level cache.

2.8 Conclusion

This work presented Contiguitas, a holistic solution across the operating system and hardware that reduces memory fragmentation due to unmovable allocations. Our evaluation at Meta’s data centers and full-system simulations showed that Contiguitas is a generic, production-ready solution to provide ample memory contiguity for not only the current demand but also a future where the optimal page size will continue to grow alongside ever-increasing memory capacities.

Chapter 3

Learned Virtual Memory: A Learned Page Table Design

3.1 Introduction

The increase in memory capacity in data centers, coupled with the proliferation of memory-intensive applications, has made virtual memory translation a major performance bottleneck. This overhead is rooted in the status quo of virtual memory translation—*radix page tables*. By design, radix page tables incur substantial overhead during page walks, requiring up to five *sequential* memory accesses.

Despite their inherent drawback of having multiple indirections, the longevity of radix page tables is justified. Organizing page tables in a tree structure (Figure 3.1(a)) provides high coverage at upper levels (e.g., 512GB of coverage at the 2nd level), with coverage diminishing only closer to the leaf nodes that map actual pages, such as 2MB at the 4th level. This approach is amenable to caching upper levels of the tree in the MMU page walk caches because applications exhibit high locality at coarse address space granularities.

Unfortunately, the meteoric rise in memory capacity has begun to pierce the veil of cacheability of radix page tables. In response, over the past decade, processor architectures have spent the last decade in an arms race to increase TLB capacity and page walk cache sizes. Despite these efforts, studies at Google and Meta reveal that roughly 20% of cycles are spent on page walks [85] (Chapter 2) and this issue is about to worsen due to several factors: (i) the inherent hardware limits of TLB scaling which have pushed latencies beyond that of L2 cache [221], (ii) the advent of terabyte-scale memory capacity through technologies like CXL [91, 182], and (iii) the increasing prevalence of memory-intensive applications.

Prior research has extensively studied methods to reduce address translation overhead [1, 3, 5, 23, 24, 25, 30, 31, 32, 33, 34, 48, 50, 63, 78, 79, 85, 98, 99, 104, 105, 107, 111, 142, 156, 158, 160, 162, 168, 180, 187, 190, 193, 194, 198, 209, 210, 228, 234] (Chapter 2). One line of work has focused on virtual and physical memory contiguity to form larger translation blocks backing application datasets [7, 25, 63, 64, 106, 107, 160, 168, 234] (Chapter 2). The goal of these works is to map large, contiguous virtual memory regions to

equally large contiguous physical memory regions and form large translations such as huge pages or even larger range translations. However, the limitation of these approaches is their reliance on abundant physical memory contiguity—a scarce resource in modern data centers (Chapter 2).

Another line of research has explored an alternative data structure to radix page tables based on hashed page tables [54, 84, 97, 100, 187, 193, 197, 228] (Figure 3.1(b)). Despite early shortcomings, recent work has shown that hashed page tables are viable [187, 193, 194, 228]. These designs replace sequential page walks with parallel accesses that aim to resolve hash collisions. However, state-of-the-art hashed page table designs [187, 193, 194] do not eliminate the core issue of multiple memory accesses for an address translation. Instead they trade sequential accesses for parallel ones, leading to increased memory traffic and cache pollution.

Recent work in databases has introduced learned indexes [52, 57, 112, 113, 132, 200, 219]. The core idea behind these structures is that the distribution of input keys can be learned, enabling models to replace hash functions and reduce collisions. Typically, learned indexes are organized as a hierarchy of models that incrementally capture finer-grained details of the key distribution. However, their effectiveness hinges on the assumption of a regular key distribution—a condition that is often challenging to satisfy in real-world database workloads [181].

In this work, we explore the design of a learned index page table structure that enables *single-access translations*. However, building page tables with existing learned indexes is challenging as they are plagued by several limitations that make them ill-suited for virtual memory translation. At a high level, learned indexes have been designed for database software and fail to meet the requirements of address translation at the processor front-end. Specifically, existing learned indexes are hierarchical and can expand significantly in both width and depth, incurring more sequential indirections than radix page tables. To achieve high accuracy, these indexes often grow to tens of megabytes and rely on floating-point arithmetic, which makes them hard to cache and unsuitable for the page walk pipeline. Additionally, existing learned indexes require large, contiguous memory allocations—a poor fit for page tables, which must function within a fragmented physical address space.

Compounding these challenges, most learned index structures are designed for static datasets and assume one-time construction. Although recent efforts have explored mechanisms to support dynamic updates [52, 57, 132, 219], these approaches either incur significant retraining overhead or suffer from accuracy degradation, ultimately resulting in performance inferior to conventional hash tables. These limitations are misaligned with the requirements of address translation, which inherently involves dynamic virtual-to-physical mappings. Moreover, supporting multiple page sizes further complicates the design, as existing techniques often require maintaining separate learned indexes per page size, undermining the efficiency gains of single-access lookups.

In this work, we holistically address these challenges through a novel learned index design tailored for virtual memory. We call our design *Learned Virtual Memory (LVM)*. At a high level, as shown in Figure 3.1(c), LVM replaces the hash functions of the hashed page table with a learned index. The learned index itself is organized as a hierarchy of

models, each responsible for different parts of the virtual address space. Intuitively, the learned index learns the location of page table entries.

We start by validating our hypothesis that virtual address spaces of applications exhibit significant regularity through an in-depth study spanning a diverse range of applications, programming languages, and memory allocation behaviors, including graph analytics, key-value stores, bioinformatics, databases, data center workloads, HPC, and real-world production workloads at Meta. Our analysis reveals that the virtual address spaces of applications are highly structured, making them well-suited for learned indexes.

To resolve the depth and width challenge of learned indexes, we introduce a cost model tailored for virtual memory. The cost model balances prediction accuracy with the cacheability of the index, accounting for the expected latency of address translation based on depth, width, and collision rates. The cost model guides LVM to build a highly optimized learned index based on space-efficient linear models on a per-process basis. The learned index of LVM requires minimal memory to provide page table entry lookup coverage for the complete address space, so it enjoys high cacheability. Next, to support efficient updates, we leverage the tendency of applications to expand their virtual address space in a contiguous manner. Specifically, in LVM, we introduce new scaling techniques for the linear models that allow new translations to be added to the learned index while avoiding retraining or rebuilding. To resolve the physical contiguity requirements, we study Meta’s production data centers and identify that while physical memory contiguity in the range of hundreds of MBs does not exist (Chapter 2), it remains abundant in the range of hundreds of KBs, even in highly fragmented servers. To this end, we design adaptive learned indexes that can dynamically allocate small leaf page tables based on the available physical contiguity. Finally, to support multiple page sizes, we develop a novel encoding scheme that leverages the linearity of our models. Different page sizes are represented via varying slopes within the same structure, enabling efficient multi-page-size support without separate indexing logic.

We evaluate LVM with OS extensions in Linux, RTL synthesis, and full-system simulations across a wide range of workloads. LVM reduces the address translation overhead by an average of 44% over radix page tables, while reducing the page walk cache area required by 1.5 $\times$. Overall, LVM achieves a 2–27% speedup in application execution time and is within 1% in terms of performance and memory traffic of an ideal page table that always performs single-access translations.

3.2 Background

In this section, we provide background on radix page tables, hashed page tables, and learned indexes.

3.2.1 Radix Page Tables

Multi-level radix page tables typically use four to five levels with 4KB leaf pages. Figure 3.1(a) shows a five-level radix page table. In the worst case, translation requires

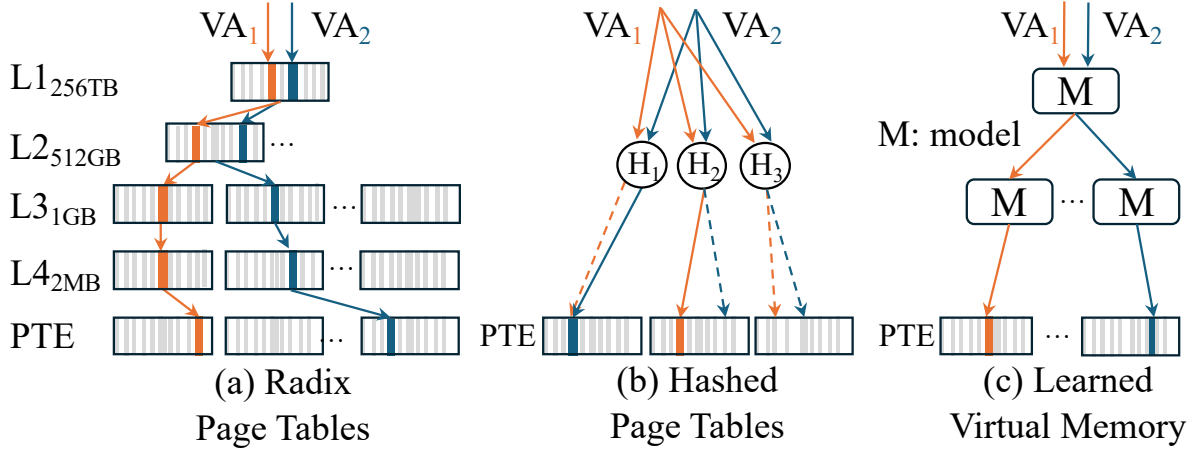


Figure 3.1: Address translations schemes. LVM avoids both the long sequential page walks of radix page tables and the high collision rates and parallel accesses of hashed page tables.

pointer-chasing through all levels, incurring five sequential memory accesses. Modern processors use two techniques to reduce translation performance overhead: large pages (e.g., 2MB) for higher TLB hit rates and shorter walks, and page walk caches (PWC) to store upper-level entries. While upper levels in radix page tables offer high early coverage, PWC reach quickly diminishes—e.g., L4 tables cover only 2MB per entry. As prior work shows [104], even with more entries, MPKI remains high due to random access patterns in memory-intensive workloads.

3.2.2 Hashed Page Tables

Hashed page tables (HPTs) [54, 70, 84, 97, 100, 187, 193, 197, 228] are an alternative to radix page tables. In HPTs (Figure 3.1(b)), a hash function maps VPNs to page table entries, enabling single-access translation in the absence of collisions. In practice, however, HPTs face several challenges [23, 228]: hash collisions can degrade performance during resolution, early designs lacked dynamic resizing and support for multiple page sizes, and scattered entries led to poor cacheability. Although clustering improved cacheability, other issues remained [228].

Elastic cuckoo page tables (ECPTs) [187, 193, 194] improve on HPTs using a dynamically resizable d -ary cuckoo hash table [61, 154] with parallel accesses. However, they still require multiple memory accesses per translation, trading sequential for parallel lookups. As shown in Figure 3.1(b), a 3-way cuckoo table probes three candidate locations, incurring two unnecessary fetches per translation [187], increasing memory traffic and cache pollution. Supporting multiple page sizes adds further overhead, as ECPT must maintain separate tables per size. While ECPTs outperform radix trees via memory-level parallelism, they still stress the memory hierarchy.

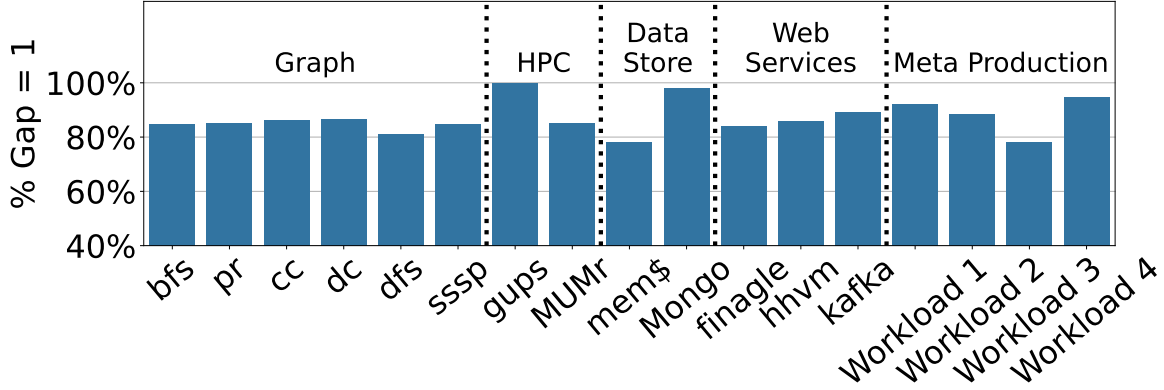


Figure 3.2: Virtual memory gap coverage of gap = 1.

3.2.3 Learned Indexes

Recent work in the database community has introduced learned indexes [113], which can outperform traditional structures like hash tables. Learned indexes replace the hash function with a hierarchy of models that approximate the cumulative distribution function (CDF) of keys, with each model responsible for part of the key space. They predict the position of the value in the table corresponding to the key, and on a misprediction, a bounded search is performed within the model’s *min* and *max* error range, leveraging sorted keys [113]. Several approaches [112, 113] have laid the foundation for learned indexes. RMI [113] uses a hierarchical structure where each node holds a simple model, reducing collisions compared to traditional hash functions [181]. A core challenge is supporting inserts: new entries reduce model accuracy and may require slow full rebuilds. Recent work [52, 57, 132, 219] enables incremental updates, but often reduces accuracy or adds complexity, increasing memory accesses during lookup.

Key Space Regularity. In practice, the primary requirement for a learned index to perform effectively is a regular key space. In this context, a regular key space is characterized by a key distribution that follows a predictable structure or pattern, e.g., sequential or evenly spaced. This regularity allows machine learning models to easily learn the distribution with high prediction accuracy.

3.3 Learned Indexes for Page Tables

In this section, we explore the feasibility of learned indexes for page tables.

3.3.1 Virtual Address Space Regularity

Since learned indexes rely on a regular key space to achieve high accuracy, we aim to determine if the application’s virtual address space exhibits a consistent pattern between its keys, i.e., virtual page numbers (VPNs). To quantify regularity, we define a metric called

the *virtual memory gap coverage*. Conceptually, a gap = 1 indicates that two virtual pages are sequential, meaning $VPN_{\text{next}} - VPN_{\text{current}} = 1$. The coverage of gap = 1 represents the proportion of all mappings in the virtual address space that are sequential. In other words, it measures how much of the virtual address space follows this specific pattern of contiguity, providing insight into the regularity of the virtual address space layout. For example, if 100% of the gaps are equal to 1, it means that all virtual pages are sequential. As a result, in that scenario, a single linear function can represent the whole virtual address space.

We use the *virtual memory gap coverage* to study the virtual address space of a broad set of representative memory-intensive applications, including graph analytics, bioinformatics, caching, HPC, database (MongoDB), web serving (Finagle RPC from Twitter [167] and hhvm from DCPPerf [174]), streaming (Apache Kafka [35]) and four production workloads at Meta (Workload 1-4). They represent applications that are written in C, C++, PHP and Java, use file-backed mappings (MongoDB) and anonymous mappings. We see that across the workloads, a minimum of 78% of gaps are equal to 1, showing significant regularity in the distribution of virtual address space of applications. Importantly, both benchmarks and real-world production workloads show similar characteristics, validating our hypothesis that the virtual address space is highly regular. To quantify the effects of different userspace allocators, we further evaluated two broadly used allocators, jemalloc [101] and tcmalloc [130]. We find that across workloads, the regularity remains practically the same.

There are several reasons why application virtual address spaces exhibit regularity. First, userspace allocators aim to minimize holes in the address space and pack allocations closely together [215]. Furthermore, allocators tend to buffer irregular allocation and free patterns of applications, hiding potential fragmentation from the OS that allocates pages for userspace [215]. Such buffering behavior is further present in virtualized environments where the guest OS handles the majority of memory allocation and free operations.

In addition, large object allocations, e.g., large arrays, need to be contiguous due to addressing requirements. Moreover, virtual address spaces are per-process and are not susceptible to fragmentation unlike the physical address space. Finally, the OS aims to reduce the space needed for page tables by coalescing mappings because lower-level page tables are only needed for the mapped virtual address space. Based on these observations, we believe that in the future, userspace allocators and the OS can make additional efforts to further bias the virtual address space towards regularity if needed, without modifications to other parts of the stack. As we show in later sections, LVM introduces cost model parameters that bound the depth and width of the learned index, further relaxing the regularity requirements of the virtual address space, even under pathological cases.

3.3.2 Virtual Memory Challenges

Having established that virtual memory is sufficiently regular to be a good candidate for learned indexes, we now discuss the requirements and challenges of an efficient virtual memory design.

Learned Index Structure and Size. Learned indexes are structured as hierarchical models that can expand in both width and depth. As a result, existing learned indexes

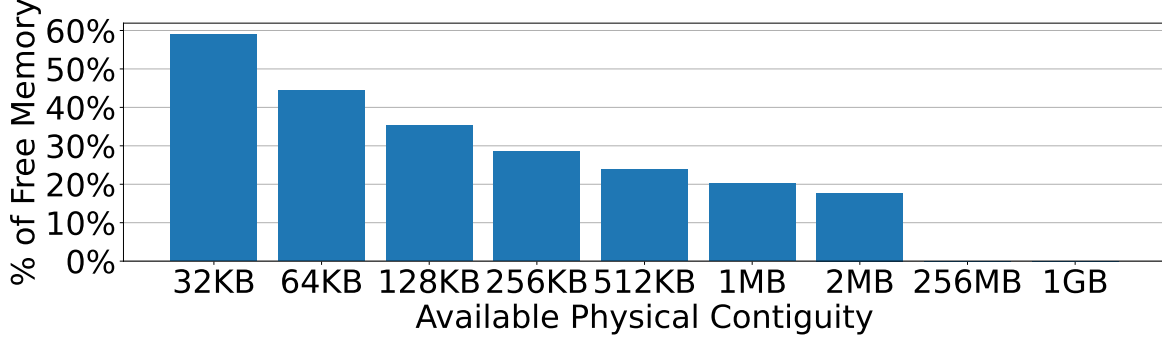


Figure 3.3: Median percentage of free memory in a Meta’s data center that can be allocated contiguously at various sizes.

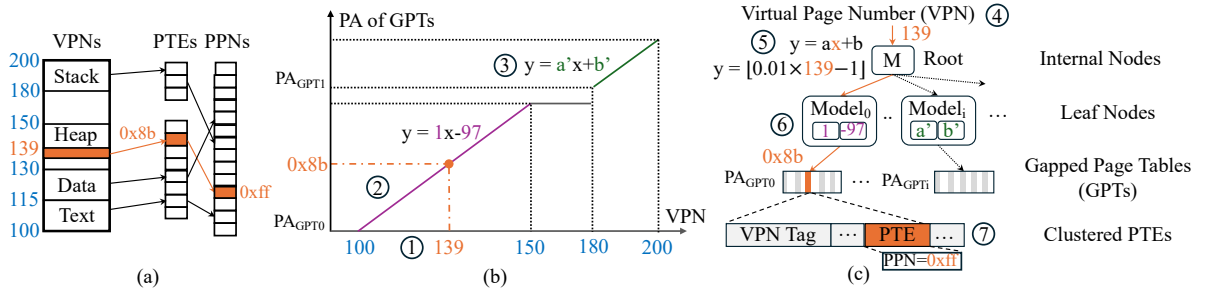


Figure 3.4: LVM learns the distribution of virtual addresses and maps them to page tables.

often involve more sequential indirections than radix page tables. Additionally, they tend to be large, frequently reaching sizes in the tens of megabytes. One potential solution is to design an MMU caching structure large enough to store these models that is loaded during context switch. However, this approach is impractical due to the significant chip area required and the added context switch overhead. Instead, to effectively use learned indexes for address translation, we need models that are compact and easily cacheable within the MMU.

Insertions and Dynamic Virtual Address Spaces. Most learned indexes are typically constructed once over a static dataset. While recent research attempts to introduce support for updates, these methods often become prohibitively expensive due to retraining or a rapid degradation in accuracy. Address translation requires learned indexes to efficiently adapt to address space changes.

Multiple Page Sizes. Virtual memory supports different page sizes, such as 4KB, 2MB, and 1GB. The page size determines the page offset and hence the size of the VPN. Prior solutions in hashed page tables [187] handle this by maintaining separate tables for each page size and auxiliary data structures providing page size information, leading to additional memory requests. Maintaining separate learned indexes for every page size negates the goal of single-access address translation. Instead, efficient translation requires a single learned index that operates with variable page sizes without loss of accuracy or increase

in training time.

Compute. Learned indexes rely on floating point operations. Implementing floating point support in the kernel, where page tables are maintained, would be challenging [58]. In hardware, while building a floating point page walk pipeline is possible, such an approach would (i) require a significant amount of resources, (ii) waste energy, and (iii) add unnecessary overhead to the page walk latency. Importantly, current MMUs support multiple outstanding page walks to satisfy the wide processor front-end; replicating such an expensive pipeline would significantly increase the area overhead. Instead, a practical page walk pipeline would require a learned index that relies on highly efficient integer computations.

Physical Memory Fragmentation. Physical memory fragmentation is a major challenge in data centers. Recent work (Chapter 2) showed obtaining large physically contiguous memory regions is infeasible in Meta’s production environments. To investigate, we conducted a large-scale study across tens of thousands of servers in Meta’s data centers. Figure 3.3 shows the percentage of free memory immediately allocatable as a contiguous block. Our findings confirm (Chapter 2): contiguous regions of hundreds of megabytes are practically nonexistent. This is a key obstacle for existing learned indexes, which typically run in software and rely on large contiguous memory for the table storing values corresponding to keys. Notably, however, we find that small contiguous regions—on the order of a few hundred kilobytes—remain widely available, even in fragmented systems. Practical learned indexes must adapt to the granularity of available physical memory to allocate their leaf page tables.

3.4 Learned Virtual Memory Design

The goal of *Learned Virtual Memory (LVM)* is to build an optimal page table structure that performs single-access address translations. To this end, LVM introduces a novel learned index design tailored to the needs of virtual memory translation.

3.4.1 Overview

Conceptually, LVM aims to replace the fixed hash functions used in hashed page tables with a learned index. LVM organizes this index as a hierarchy of models, where each model is responsible for a subset of the virtual address space. Intuitively, the learned index predicts the location of page table entries (PTEs). Figure 3.4 provides a high-level overview of how LVM learns the structure of the virtual address space and maps it to page table entries.

Starting with Figure 3.4(a), each application has its own virtual address space, containing segments such as text, data, and heap. As in radix page tables, each mapped virtual page number (VPN) has a corresponding PTE, which specifies the physical page number (PPN). In radix page tables, the VPN-to-PTE mapping is performed through a fixed number of intermediate page tables that are indexed with a subset of the VPN bits. In hashed page tables, the mapping is identified by passing the VPN through a fixed hash function.

Intuitively, LVM replaces the fixed hash function with a learned function based on the underlying structure of the virtual address space. To showcase this point, we next visualize the mapping of VPNs to PTEs in Figure 3.4(b). In this figure, the x -axis represents the sorted VPNs, while the y -axis shows the corresponding physical addresses of PTEs. The goal of LVM’s learned index is to discover a set of functions that, given a specific VPN as input x , compute an output y that accurately predicts the physical address of its corresponding PTE. This concept is central to how LVM operates—it learns the distribution of VPNs and stores the PTEs at locations defined by the learned function.

LVM realizes this idea through a hierarchy of models, as shown in Figure 3.4(c), which partitions the virtual address space into smaller subsets that are described by models in *internal* and *leaf* nodes. Intuitively, internal nodes zoom in on a particular subset of the address space. They take as input the VPN and output the location of the child node responsible for an even smaller piece of the address space. Leaf nodes are responsible for the final mapping to page table entries and take as input the VPN and output the physical address of the PTE. Internally, the operation of both internal and leaf nodes is similar. Specifically, since the virtual address space is highly regular (Section 3.3.1), LVM leverages simple linear functions that effectively describe the distribution of the address space shown in Figure 3.4(b). LVM stores PTEs in gapped page tables (GPTs), which are conceptually small arrays of page table entries. Each GPT can be allocated at different physical address ranges, and hence, LVM does not impose strict physical contiguity requirements.

Under LVM, the OS maintains the learned index and the gapped page tables, and the hardware, on a TLB miss, performs lookups to locate the PTE. As we will discuss later, LVM aims to optimize the learned index’s structure—including depth, width, and other parameters—to improve address translation efficiency.

An Address Translation Example with LVM. Next, we describe an end-to-end translation example with LVM. Consider a page in the application’s heap with a $VPN = 139$, as shown in Figure 3.4(a). This VPN is mapped by a PTE that is located in the PA $0x8b$. The PTE maps that page to PPN $0xff$.

Figure 3.4(b) shows the VPNs of the application in the x -axis, and ① shows VPN 139. Based on the distribution of VPNs shown in the figure, LVM learns a representation of the mapped pages using the linear function $y = 1 \cdot x - 97$ as shown in ②. Specifically, to identify the location of the PTE, we set x as the VPN under translation, and the output y is the physical location of the gapped page table. Similarly, the pages in the application’s stack are represented by another linear function (③ in Figure 3.4(b)).

Putting everything together, Figure 3.4(c) illustrates a page walk for $VPN = 139$. The walk begins at the root node (④), which spans the entire virtual address space of the application. The root contains a linear model $y = ax + b$ that uses the input $x = 139$ to select one of its child nodes, as shown in step ⑤. In this case, the root has a learned function $y = 0.01x - 1$ to select between its children nodes. The round-down result of this function with $x = 139$ directs us to $Model_0$, which is responsible for the VPN range $[100, 150)$. At step ⑥, the selected model computes $y = 1x - 97$ for $x = 139$, returning the physical address $0x8b$. This address corresponds to the location of the target page table entry (PTE) within the per-leaf-node gapped page table starting at base address PA_{GPT0} . Next, the relevant PTE cluster is fetched from memory (step ⑦), and the cluster’s VPN

tag is checked to validate the entry. Finally, the retrieved PTE contains the physical page number (PPN) = 0xff.

3.4.2 A Learned Index for Virtual Memory

In this section, we describe the learned index structure, the underlying gapped page table, and the cost model of LVM.

The Learned Index Structure

LVM introduces a novel learned index design specialized to serve the needs of address translation. The purpose of the learned index is to identify the location of the PTEs. The learned index is organized as a hierarchy of models, each residing logically within a node. This is because a single model is ineffective at capturing the complete distribution of the keys. Figure 3.4(c) shows the overview of LVM’s learned index structure. There are two types of nodes: internal nodes subdivide the key space into finer granularities to be handled by their children, and leaf nodes predict the position of the value associated with a key in the table. The children nodes evenly divide the address space of a parent node. The number of children per node is flexible and depends on the complexity of the key space. This approach enables a pliable index that can adapt to the needs of different applications. As we will discuss in Section 3.4.2, the training process is responsible for defining the depth and width of the hierarchy.

The learned model is the building block of the index. There is an inherent relationship between the complexity of the model and the address space it aims to represent. As we saw in Section 3.3.1, the virtual address space of applications is highly regular. Consequently, LVM opts for a linear function of the form $y = ax + b$. Hence, each node only needs to store the slope a and the intercept b . This design decision offers ease of computation (one multiplication and one addition) and provides sufficient accuracy when used in a hierarchy. Furthermore, it greatly enhances the explainability of the model.

Each model is responsible for approximating the cumulative distribution function (CDF) of its key space and mapping its key space to its output range, which is either the number of child nodes (for an internal node) or the size of the table (for a leaf node). Essentially, each model outputs $range \cdot CDF(x)$, where x is the key.

LVM uses this learned index design for address translation. Specifically, the key space is the virtual address space of the application. The input key is the virtual page number (VPN) under translation. The output of the leaf node is used to access the gapped page table holding the PTE, as shown at the bottom of Figure 3.4(c).

LVM stores all internal nodes at a particular depth consecutively in physical memory. Hence, each internal node can be uniquely identified with an offset, obviating the need to output the physical addresses of the internal child nodes and significantly reducing the size of each node. This layout does not impose any meaningful physical memory contiguity requirements as the learned index models of LVM are tiny, which we will see in Section 3.7. **Scalability.** Because a single function can describe an arbitrarily large, virtually contiguous space, the learned index remains efficient even as memory usage grows. In contrast,

radix and hashed page table designs require linearly larger MMU caches. For example, radix page tables need to populate additional entries in the upper levels of the radix tree as the memory footprint increases. This leads to cacheability issues since the hardware page walk caches (PWCs) must also expand—lest additional sequential memory accesses be needed to traverse the page tables. In contrast, LVM provides long-term scalability and cacheability in hardware irrespective of the memory footprint, effectively future-proofing its design. We empirically demonstrate this property of LVM in Section 3.7.

Gapped Page Tables

Each leaf node in LVM is associated with its own page table. The model within the leaf node maps the VPNs to PTEs. To support insertions after the model is built, the page table is organized as a *gapped array* [52], resulting in *gapped page tables* (Figure 3.4(c)). Intuitively, the gapped page table contains additional empty slots, called gaps, that are unoccupied at build time. This design allows new keys to be inserted and further reduces collision rates when combined with LVM’s insertion procedure, which we discuss in Section 3.4.3.

Relaxing Physical Contiguity Requirements. An important design decision for the gapped page tables is their size. This is because gapped page tables, similar to regular page tables in radix or hashed page tables, are allocated in the physical address space. As we showed in Figure 3.3, large blocks of physical contiguity are scarce in real-world data centers due to fragmentation. Instead, LVM optimizes the design of the gapped page tables for small allocations that are ample even in highly fragmented environments and eliminates the need for a large physically contiguous page table. To this end, LVM maintains different gapped page tables for each leaf node as shown in Figure 3.4(c) with different physical addresses. To achieve this, during training of a leaf node, the physical address of the base of the gapped page table is added to the index of the PTE in the table to yield the final output that the leaf model needs to learn. In the example in Figure 3.4, this means adding PA_{GPT0} to the index of the PTE in the GPT, using it as the output the leaf model learns. Therefore, leaf nodes in LVM output directly the physical addresses of PTEs in its private table.

Moreover, this per-leaf-node gapped page table design enables LVM to dynamically adapt to the physical memory fragmentation status of the system when a leaf node is created. If there is insufficient physical contiguity for a large table, additional leaf nodes—and thus separate gapped page tables—can be created to utilize smaller physical contiguity as low as the base page size, e.g., 4KB in x86 or 64KB for some ARM systems. In the evaluation, we will demonstrate the adaptability of LVM and show that this approach performs well even in highly fragmented environments.

Cost Model

The shape of the learned index is critical for performance and space efficiency. There are three key parameters that influence the lookup performance of learned indexes in the context of virtual memory: (i) the depth of the index, which defines the number of indirections required to traverse the model, (ii) the branching factor of the nodes, which

influences the width of the model and affects the translation coverage of each node, and (iii) the collision rate, which determines the number of accesses required to locate an entry in the underlying table. Generally, increasing the depth or branching factor allows individual models to handle smaller segments of the virtual address space, which decreases collision rates and enhances lookup performance. However, as the index becomes wider and deeper, traversal times increase, and the model’s cacheability deteriorates due to its larger size.

To address this trade-off, LVM introduces a principled cost model to balance these parameters effectively. The cost model employs a weighted function associated with a specific number of children (n) for each node under consideration. This function incorporates the index depth (d), the size of the index in bytes (s), the collision rate (cr), and the average number of additional memory accesses per collision (ma). By optimizing this function, LVM finds an equilibrium that minimizes overall lookup latency while maintaining efficient space utilization. Specifically, we define the translation cost $C(n)$ of a node with n children as:

$$C(n) = x_1 \cdot d + x_2 \cdot s + x_3 \cdot cr \cdot ma \quad (3.1)$$

We discuss the tunable weights x_i in Section 3.5. During execution, the cost model first computes the number of *spline points* for the keys within a particular node [112]. Spline points partition the input key space into segments where simple linear models can accurately approximate the key distribution. Essentially, the number of spline points reflects the complexity of the key distribution within the node—more spline points indicate greater variability. This number provides a reasonable estimate for the optimal number of child nodes needed to model the key space effectively. To reduce computational overhead, the cost model limits its evaluation to ± 2 around the estimated number of spline points, allowing the cost model to quickly assess a few plausible numbers of child nodes, minimizing the effort required to find the most cost-effective option.

Although the cost model inherently penalizes increasing the index depth without significant accuracy gains, LVM further enforces a hard limit on the depth using a parameter `d.limit`, which bounds the number of indirections during a lookup, ensuring efficient traversal. Additionally, LVM imposes a constraint on the worst-case coverage per byte, aiming to match or improve upon the locality provided by a radix page table at the same level. Specifically, if a node does not offer sufficient coverage relative to its size—that is, it doesn’t map enough of the address space per byte of index—the cost model prevents the creation of additional child nodes beneath it. This strategy emphasizes the cacheability of the index by ensuring that each node is space-efficient. By bounding the depth and coverage of the learned index, these two constraints also protect against pathological cases when the regularity of the virtual address space is not maintained. Even in a pathological case, LVM’s learned index would not grow too deep for hardware page walks nor too large to have good cacheability.

LVM’s cost model is carefully tailored to the needs of virtual memory systems. Conceptually, the goal is to create a more efficient hierarchy than the upper layers of a radix tree. In practice, while a radix tree has a rigid structure, LVM’s cost model enables a flexible hierarchy that can adjust its depth and width to increase precision for specific regions of the address space. The model dynamically adds depth and width by closely following

the regularity of the address space, incorporating additional nodes only where they are necessary. As we will demonstrate in Section 3.7, this cost model leads to indexes that are minimal in size yet highly efficient.

3.4.3 Learned Index Operations

In this section, we discuss the main operations of LVM.

Initialization

The learned index is initialized as follows. Given a set of keys, LVM creates a root node and begins the training process. This training invokes a cost model that greedily determines the optimal number of child nodes at each level. When a new branch is created, the key space is evenly divided into k segments, assigning each child node responsibility for $1/k$ of the key space. This recursive process continues for each child node until the cost model decides that a leaf node should not have any further subdivisions. The same procedure is applied when rebuilding the model. In the context of virtual memory, the operating system initializes the index upon mapping the first page of a process.

Training of the Learned Index

The goal of training the linear model in a node is to approximate the cumulative distribution function (CDF) of the keys as closely as possible with minimal computational cost. Internal nodes and leaf nodes are trained using two different methods. For internal nodes of the index, the training process first determines the number of children the node should have based on LVM’s cost model. Then, the training continues by allocating memory space for the child nodes. Next, to derive the model of the internal node, LVM learns a linear function between VPNs and indexes of children nodes to evenly divide the parent node’s key space. Because the relationship between inputs and outputs can be perfectly represented by a linear function, heavyweight algorithms are unnecessary.

Leaf nodes are trained differently through a specialized process. First, LVM queries the OS allocator for physical contiguity (e.g., the next available allocation order in the buddy allocator in Linux). Then based on the available contiguity, LVM adapts the number of sibling leaf nodes necessary such that a leaf node covers as many translations as allowed by physical contiguity. A linear model for a leaf node is then learned using linear regression over the (VPN, position) pairs, where position denotes the index of the VPN in the sorted list of all VPNs within that node. This model is then scaled by the gapped array scale factor (`ga_scale`) to disperse the existing VPNs across a gapped array, leaving empty slots available for future insertions. After scaling, a table for the node is allocated in memory with size determined by multiplying the total number of VPNs at the node by `ga_scale`, ensuring sufficient space for both current and future VPNs. The page table entries corresponding to the VPNs are then inserted into the gapped page table array at the positions predicted by the scaled model. If a predicted location is already occupied, an exponential search is conducted to find the next available slot for insertion.

Bounding Mispredictions

As learned indexes are based on machine learning models, they can incur rare mispredictions during a lookup. If the predicted location does not result in a hit, then a binary search is performed within the error bound that is enforced during the training of the index [113]. For LVM, this is done by setting the error bound during linear regression such that an upper bound C_{err} of the number of additional memory accesses is enforced to ensure efficient address translation even in the worst case. This approach sacrifices some space efficiency for tighter error bounds by creating more child nodes during training. More specifically, if, at a particular leaf node, the error bound cannot be satisfied, LVM goes back to its parent node and re-evaluates the cost model with a sufficiently boosted weight x_3 for the computational cost of resolving a collision that will satisfy the error bound in all its children nodes. Training then returns to the parent node, and more child nodes are created to subdivide the key space at a finer granularity, which simplifies the key space each child node is responsible for and reduces error.

Supporting Efficient Insertion

Insertions are divided into two categories based on the new key’s relation to the existing key range: (i) within-bounds inserts and (ii) out-of-bounds inserts. A within-bounds insert occurs when the new key falls within the existing range of keys, while an out-of-bounds insert happens when the new key lies outside the current key range.

Out of bounds inserts close to the edge. LVM optimizes for the common case of out of bound inserts that happen close to the existing edge. This is because applications tend to exhibit locality in page allocations, so newly added pages tend to expand the virtual address space in a contiguous manner. LVM introduces two new novel techniques for learned indexes to avoid costly retraining.

The first technique defines the *minimum insertion distance* which allows insertions to be processed in batches. Specifically, LVM always expands the address space by at least the minimum insertion distance. The benefits are two-fold. First, this approach absorbs some non-contiguous insertions that are temporary in nature. Second, it allows LVM to amortize the fixed computational cost of insertions over a larger range of keys, further reducing the software management cost. Note that LVM’s approach of allocating a minimum distance only creates page tables ahead of time, and the actual physical page allocation only happens later on demand. The second technique relies on rescaling. Specifically, LVM rescales the leaf node by expanding the gapped page table associated with the node without modifying the model and inserting the new keys.

For example, as shown in Figure 3.5 (a), suppose a leaf node (Model_0) is responsible for keys 500 – 1000 and has its gapped page table. During an insertion of a new VPN, 1030, first, LVM records that the key range of this node is expanded to 500 – 1050, taking into account a hypothetical minimum insertion distance of 50 (Step ① in Figure 3.5 (b)). Then, the gapped page table is expanded by 65 slots (Step ②), taking into account $\text{ga_scale} = 1.3$. Finally, the new key for VPN 1030, along with other keys from 1001 – 1050 are inserted into the expanded part of the gapped page table at locations predicted by the

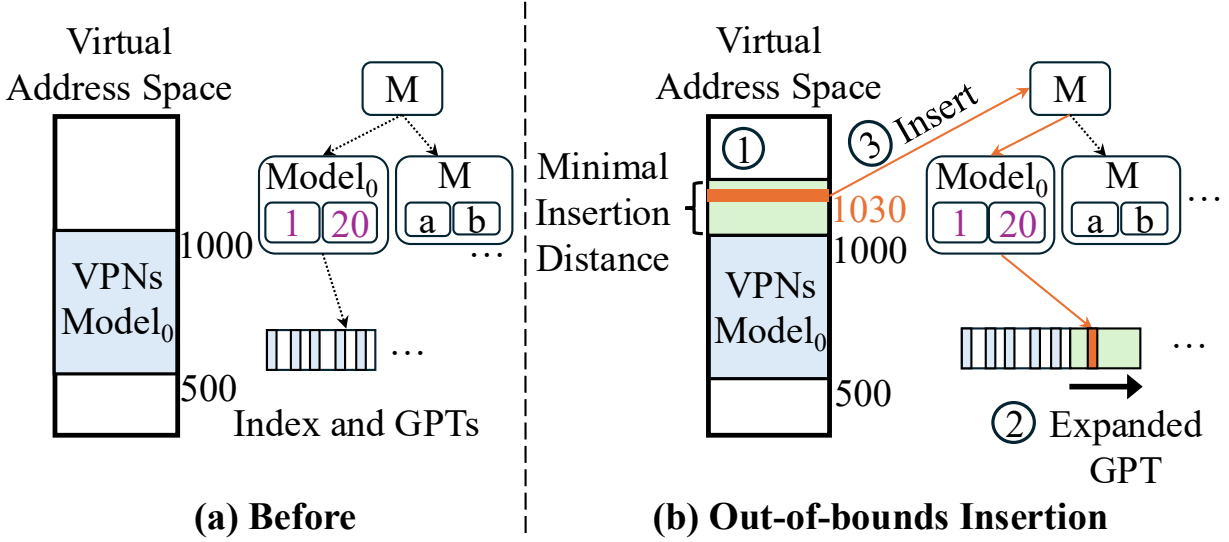


Figure 3.5: LVM out of bounds inserts close to the edge.

unchanged model (Step ③). Note that this avoids retraining the model or re-inserting the existing keys of the node: for existing keys for VPN 500 – 1000, since neither the model nor the location of their PTEs changed, future page walks still yield the correct PTEs; similarly, the expanded keys follow the same model. Overall, these techniques, when combined with the underlying design of gapped page tables, resolve the adaptability limitation of prior learned indexes, making them suitable for virtual memory.

Within bounds inserts and out of bounds inserts away from the edge. If the key to be inserted falls within the current bounds of the key range, LVM proceeds as follows. LVM queries the model and attempts to insert the key into the location indicated by the model. Due to LVM’s gapped array design, the table usually has empty slots for insertion. In the common case, the location is empty, and the new key is inserted successfully. Otherwise, in the unlikely scenario that the slot is occupied, LVM proceeds with retraining only the leaf node. In the vast majority of cases, the local retraining is successful, and the new key finds its place within the table.

Finally, if the key is still not inserted, there are two possible approaches we can take. These approaches are also the same for out of bounds inserts away from the edge. One approach is to only rebuild the parent node and try again. This process can continue recursively until the key is inserted. The other approach is to simply rebuild the index. In practice, as we will see in Section 3.7, the computational cost of rebuilding the model is quite small. Hence, due to the rarity of these events, LVM opts for a full rebuild.

3.4.4 Supporting Multiple Page Sizes

LVM efficiently supports multiple page sizes without requiring separate learned index structures, thereby overcoming the limitations of hashed page table designs. It leverages the linearity of the learned index to seamlessly accommodate multiple page sizes within a single index. This is achieved by representing different page sizes as lines with varying slopes in

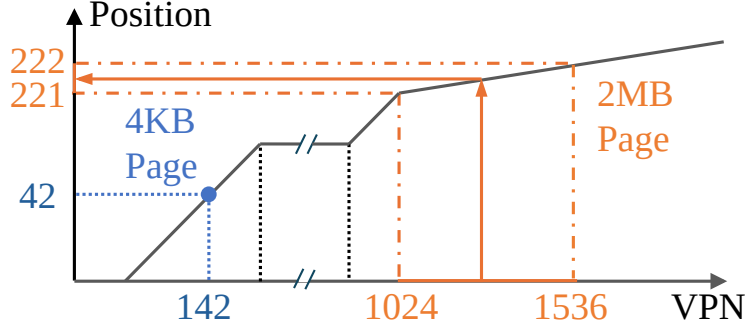


Figure 3.6: Regular and huge pages as represented by LVM.

the cumulative distribution function (CDF) of the keys. Specifically, smaller pages correspond to steeper slopes, while larger pages correspond to lower slopes. Intuitively, larger page sizes result in fewer PTEs within a given VPN range, leading to lower slopes in the learned index.

Figure 3.6 shows an example virtual address space distribution with both a 4KB page and a 2MB page. The 4KB page at VPN 142 has only one VPN in its range of virtual addresses, which gets mapped to position 42 in the gapped page tables by the learned index. On the other hand, the 2MB page spans VPNs [1024, 1536). To support huge pages, LVM trains the learned index based on the VPN of the first 4KB sub-page i.e., VPN 1024. During a lookup, the 4KB VPN is derived from the VA to traverse the index. Note that if a huge page is mapped, 4KB sub-pages inside it cannot also be mapped. Therefore, lookups using any addresses inside a huge page will identify a unique translation entry. In the example of Figure 3.6, any queries using VPNs between 1024 and 1536 will be rounded down to the PTE at position 221, yielding the 2MB translation entry.

To identify the page size, the translation entry in the table uses two bits to encode the size (supporting the 4KB, 2MB, and 1GB pages in x86). Importantly, LVM can support *any* number of page sizes without modifications or additional structures. This is in contrast to existing radix page tables that rely on rigid page sizes and hashed page tables [187] that require separate structures and potentially additional accesses for each page size. We envision that LVM will enable a new direction where applications can leverage arbitrary page sizes without hardware changes, unlocking high performance. We leave this exploration for future work.

3.4.5 Fixed-Point Arithmetic for Fast Lookups

State-of-the-art learned indexes rely on floating-point operations, which violates the requirements discussed in Section 3.3.2. To ease the implementation of LVM, we introduce an effective quantization strategy for learned indexes that represents model parameters in fixed-point values. Specifically, each value in the model consists of a 44-bit integer part and a 20-bit fractional part. In this way, each value of a model requires 8 bytes, and each node takes 16 bytes.

3.4.6 OS and Hardware Support

At a high level, LVM maintains the same abstraction and mechanisms of virtual memory as in radix and hashed page tables.

Operating System

The OS components of LVM build and manage per-process learned indexes. They include the cost model along with logic in the page fault handler and generic page walk logic for the OS to access and modify translations. In terms of operations, the OS is responsible for initiating, training, and manipulating the learned index and the translations, while the hardware performs lookups during address translation. We further discuss our OS prototype of LVM in Section 3.5.

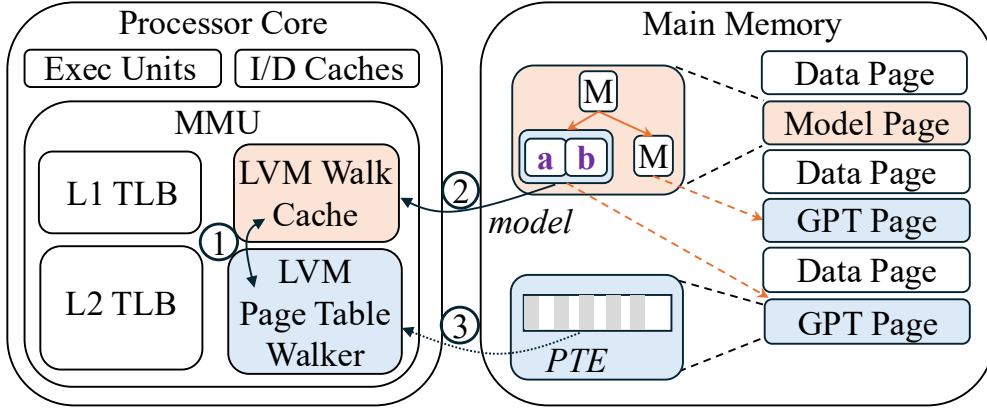


Figure 3.7: LVM hardware overview.

Hardware

The changes of LVM are isolated in the hardware memory management unit (MMU) as depicted in Figure 3.7. Only the two shaded components in the MMU are modified compared to existing architectures, while other components, such as the L1/L2 TLBs, remain unchanged. The page table walker and the page walk cache (PWC) of the radix page table are replaced by the *LVM Page Table Walker* and the *LVM Walk Cache (LWC)*. Each LVM walker contains an adder and a multiplier that are used to compute the output of the model. Upon an L2 TLB miss, the LVM page table walker starts to translate a virtual address to a physical address. It performs the LVM page walk by traversing through a hierarchy of learned models, which are cached by the *LVM Walk Cache (LWC)*. If a requested model is present in LWC, it is supplied to the page table walker (Step ①); otherwise, it is fetched from main memory on demand (Step ②). After the page walk reaches the leaf node in the index, the physical address of the page table entry (PTE) is computed, and

the walker fetches it from the memory hierarchy (Step ③). Similarly to radix, the PTEs can be cached by the data caches of the processor, but are not cached in the LWC.

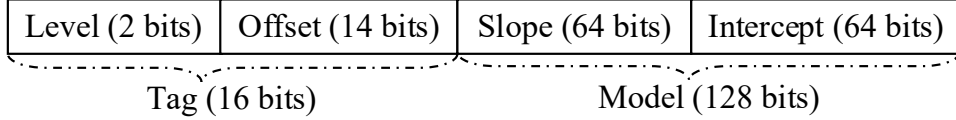


Figure 3.8: LVM Page Walk Cache Entry.

The LWC is fully associative and stores the slope and intercept of individual models of the learned index. Figure 3.8 shows an LWC entry in details. This design allows LVM to only cache nodes on demand instead of fetching the whole learned index. The page table walker fetches 64-byte cache lines, each containing four individual models. Each model takes 16 bytes of space (slope and intercept). The page walker stores the learned index node in an LWC entry and attaches the level and the offset of the node. Furthermore, each entry in the LWC stores the address space identifier (ASID). This approach allows LVM to handle context switches without flushes, similar to PWCs in radix page tables. As we will see in Section 3.7, LVM achieves a $3\times$ area reduction over radix page walk caches while being more scalable.

The OS exposes information required for the hardware page walk using `d_limit` control registers (Section 3.4). LVM adds these registers that store the base physical address of the internal nodes at each level in the index. This is similar to how the x86-64 architecture stores the base address of the page table in the CR3 register.

Virtualization Support. LVM can support virtualization through nested page tables, similarly to radix page tables. In this environment, the hypervisor maintains LVM page tables for each VM to translate guest physical addresses (GPA) to host physical addresses (HPA). Similarly, each guest OS maintains LVM page tables to translate guest virtual addresses (GVA) to guest physical addresses (GPA). Due to the increased performance cost of nested radix page tables, we expect LVM to provide even higher performance gains.

3.5 Implementation

3.5.1 Parameters Used

LVM uses a set of tunable parameters. We empirically set the values of the tunable weights in the cost model to be $x_1 = 10, x_2 = 5, x_3 = 200$. We define the depth limit of the index `d_limit` = 3 to mirror the maximum number of indirections in a traditional radix page table (LVM has another fetch for the translation entry, bringing the max possible memory accesses to 4). We set `ga_scale` = 1.3 to effectively leave space for future inserts in the table. Finally, we set the *minimum insertion distance* to be 64MB, enough to cover most inserts in applications and not unduly expand the size of the table.

3.5.2 LVM Operations in the OS

Software lookup. When the translation entry of an already mapped page needs to be changed or looked up (e.g., a permission change or checking the *accessed* or *dirty* bits), the OS performs the same page walk operation as the hardware does to obtain the page table entry. These operations do not modify the index.

Free. When a page is freed to the OS, LVM follows the same steps as radix page tables by clearing the PTE in page tables, flushing the TLB, and releasing the physical page. However, LVM opts to not change the index and keeps the empty space for the PTE in the table because, first, the OS prefers to keep the virtual address space contiguous and plugs in holes in the virtual address space with later allocations, so the space for the entry will likely be reused in the future by new allocations. Second, memory usage of applications typically remains relatively stable near their peak usage after their initialization phase. For workloads that exhibit a peak memory usage that is significantly higher than their steady-state memory usage, the OS can rebuild the index and reclaim unused space.

Insert. LVM maps new pages by inserting into the learned page table as described in detail in Section 3.4. Physical frame allocations are deferred until the first access to a page, as in current OSs.

Kernel Mappings. The Linux kernel has a shared kernel address space mapped to the virtual address space of all processes. LVM keeps one learned page table for the kernel address space, reducing memory consumption and also avoiding repetitively training a different kernel space learned page table for each new process.

ASLR. Address space layout randomization (ASLR) partitions the virtual address space into distinct parts that are far apart. The OS exposes the ASLR base addresses to hardware through registers, removing ASLR effects during LVM training and management. This approach is transparent to applications, maintains ASLR’s security, and simplifies the learned index training process of LVM.

TLB Shootdowns. LVM maintains the same TLB shutdown behavior as radix page tables. Specifically, TLB shutdowns take place only when PTEs are modified, e.g., during OS swapping or page permission changes. During any LVM-specific operation, such as retraining, insertions, or rescaling, PTEs are not modified and hence a TLB shutdown is not required.

LWC Flushes. On regular insertions or rescaling, which are the most frequent operations, the LVM learned index is not modified and hence the LWC does not need to be flushed. On the very rare occasion when an LVM model node is retrained, it causes the cached entry in the LWC to become stale. In this case, the OS simply flushes the entry from the LWC. As we will see in the evaluation (Section 3.7.3), retraining operations, and thus LWC flushes, are very rare, with an average of two and at most three occurring during the entire application runtime, taking place at the program initialization.

Multi-threading LVM naturally supports multi-threaded applications. At a high level, LVM leverages similar locking mechanisms as in radix page tables. For updates to the PTEs, the table in which the PTE resides is locked by the OS so that only one thread can perform the modification at any given time. This locking requirement is consistent with PTEs being modified in radix page tables, when the OS acquires a lock associated with the

leaf page table. For retraining, LVM acquires a lock on the parent node during retraining, preventing concurrent modifications. This is similar to when radix page tables allocate or free page tables.

3.5.3 Linux Prototype

We evaluate LVM with a Linux prototype. Specifically, we modify Linux kernel 5.15 to stream operations that map and unmap pages to a userspace agent that implements LVM. As the hardware features of LVM are simulated, we choose to implement LVM in userspace through an indirection layer for ease of development. Our prototype maintains LVM for a particular process, including all the necessary virtual memory operations, building and training the learned index, as well as managing the translation tables, including allocation and free operations. During simulation, we can configure our prototype to rely solely on LVM for virtual memory operations, providing a realistic evaluation of its performance. The prototype is composed of 4200 lines of C and C++.

3.6 Evaluation Methodology

3.6.1 Architectural Simulation

We use SST [175], a detailed cycle-level simulation backend, integrated with a QEMU [29] frontend that runs the Linux kernel and userspace processes for full-system simulation. SST also incorporates DRAMSim3 [178] for main memory modeling. We model all the hardware components of LVM, including page walkers, LWCs, and flush operations. The architectural parameters are shown in Table 3.1. The simulation includes all the computational costs of managing LVM, e.g., training, insertions, and rescaling by relying on our Linux prototype described in Section 3.5.3.

3.6.2 Workloads

We consider a wide range of workloads in our evaluation. They include six workloads from the graphBIG [151] graph benchmark suite: Breadth First Search (BFS), Depth First Search (DFS), Connected Components (CC), Degree Centrality (DC), Page Rank (PR), and Shortest Path (SSSP). They take a Kronecker graph that produces a runtime memory footprint of 75GB. GUPS is a random access benchmark from HPC Challenge [133]. MUMmer (MUMr) is a DNA sequence alignment program from the BioBench2 [6] bioinformatics suite and has a memory footprint of 20GB. Memcached [60] (mem\$) is a widely used in-memory key-value store. It has a memory footprint of 124GB. During simulation, we execute 1 billion instructions in the region of interest of the benchmarks.

3.6.3 Configurations and Baselines

We compare **LVM** with four-level radix page tables and elastic cuckoo page tables (**ECPT**) [187], a state-of-the-art hashed page table. We model both designs in detail, including a page

Full-System Simulation Parameters	
Core	4-issue out-of-order cores at 2GHz
L1-I and L1-D cache	32KB each, 8-way, 1 cycle RT
L2 cache	1MB, 8-way, 20 cycles RT
L3 cache	2MB per core, 16-way, 56 cycles RT
L1 DTLB/ITLB (4KB pages)	64 entries, 4-way
L1 DTLB/ITLB (2MB pages)	32 entries, 4-way
L2 TLB (4KB pages)	2048 entries, 12-way
L2 TLB (2MB pages)	2048 entries, 12-way
Radix Page Walk Cache	3 levels, 32 entries per level, 2 cycles
LVM Page Walk Cache	16 entries, 2 cycles
Cuckoo Walk Cache	PMD: 16 entries. PUD: 2 entries. 2 cycles
Cuckoo Page Tables	3 ways. 16384 entry initial size.
Main Memory	128GB, DDR4 3200MT/s, 8 banks, 4 channels
OS	Linux Kernel 5.15

Table 3.1: LVM architectural parameters.

walk cache (PWC) for radix page tables, and for ECPT, we include the Cuckoo Walk Cache (CWC) and Cuckoo Walk Tables (CWT) that trim the required parallel accesses for huge pages. The configuration parameters are shown in Table 3.1. Furthermore, to get a better understanding of the upper bound performance benefits, we compare LVM to an **Ideal** page table scheme that always requires only a single memory access to locate the page table entry to complete a page walk. For every configuration, we consider two page sizes: (i) **4KB**, and (ii) 2MB using Linux’s Transparent Huge Pages (**THP**).

3.7 Evaluation

In the evaluation, we first present the end-to-end speedups of workloads in Section 3.7.1. We also compare the performance of LVM to three related work, Midgard [76], ASAP [142] and FPT [158]. Next, to understand the sources of the speedups, we investigate in Section 3.7.2 the architectural characteristics of LVM regarding MMU, caches and the memory hierarchy. Then, we characterize the learned page table data structures of LVM in Section 3.7.3 in terms of size, collision rates, memory fragmentation, memory consumption, etc. Finally, we evaluate the hardware properties of LVM in Section 3.7.4 using our implementation and synthesis in RTL and a commercial PDK and compare its area and power to those of Radix Page Tables.

3.7.1 End-to-end Speedups.

In Figure 3.9, we show the end-to-end performance results. We report speedups relative to the 4KB radix page tables. We see that LVM provides major performance gains. When using 4KB pages, LVM achieves a 5%–26% speedup over radix page tables, with an average

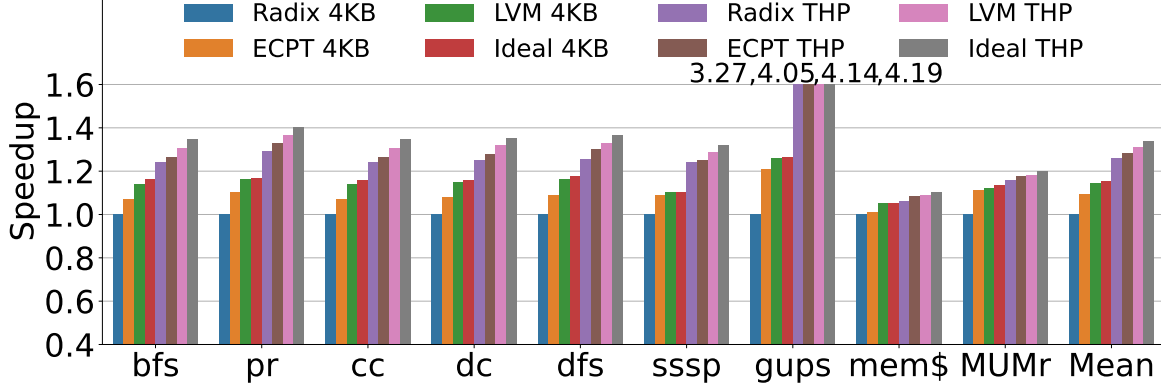


Figure 3.9: End-to-end speedups of various page table designs.

speedup of 14%. For THP, LVM achieves a 2%–27% speedup over radix page tables, with the average speedup being 7%. When compared to ECPT with 4KB, the state-of-the-art hashed page table design, LVM achieves up to a 7% speedup, with an average speedup of 5%. These are significant performance gains. Similarly, LVM outperforms ECPT by 3% when using huge pages. As we will see next, ECPT achieves good performance by trading latency for increased memory traffic due to parallel accesses. LVM outperforms ECPT and drastically reduces memory traffic for address translation.

Next, to get a better understanding of the upper bound performance of LVM, we compare it to an ideal design that always requires only a single memory access for locating the page table entry. LVM achieves performance within 1% of an ideal page table design on average. Overall, LVM provides significant speedups over radix page tables and state-of-the-art hashed page table designs, providing near-ideal address translation performance. **Multi-tenancy.** To evaluate the efficacy of LVM in multi-tenant environments with stacked workloads, we performed additional simulations with combinations of our workloads on an 8-core setup, and every workload runs on its own core. The speedups of LVM remain the same (within 0.5% difference) across configurations.

Multi-threaded Applications. LVM supports multi-threaded applications as described in Section 3.5.2. We simulated the graph workloads with eight threads. LVM maintains its performance gains and the results remain similar (within 1% difference) across configurations. This is because LVM efficiently uses fine-grained locking for page table modifications and introduces mechanisms (rescaling and minimum insertion distance) that make retrains exceedingly rare (Section 3.7.3).

3.7.2 Architectural Characterizations.

MMU Overhead. To understand the performance results, we look at the MMU overhead relative to radix page tables shown in Figure 3.10. We measure the total cycles that memory requests spend in the MMU, including in the TLB and page table walker. Note that for the 4KB configurations, we normalize the results to Radix with 4KB pages and the THP

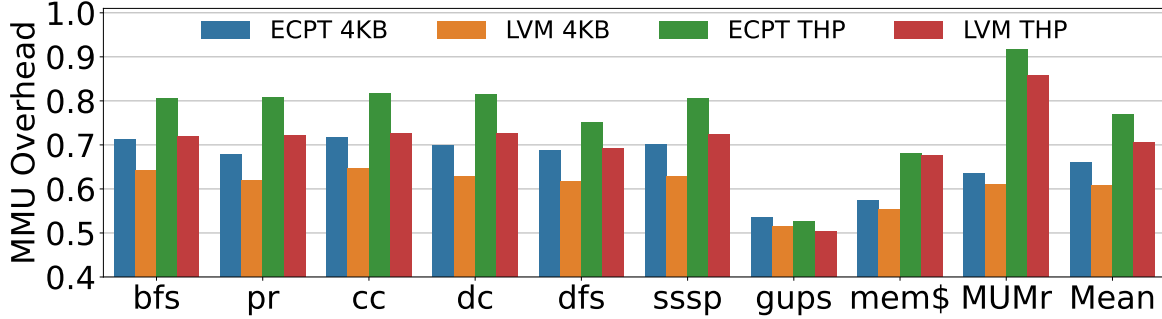


Figure 3.10: MMU overhead relative to radix. Results are normalized separately to radix 4KB and THP.

configurations to Radix with THP. As we can see from the results, LVM reduces the MMU overhead by an average of 39% compared to Radix for 4KB pages and by 29% when using THP. Furthermore, LVM outperforms ECPT by 3%–10% and 8% on average for 4KB pages and by 1%–11% and 8% on average under THP. In terms of page walk cycles, LVM achieves a 52% reduction when using 4KB pages and 44% reduction when using THP on average compared to radix. In comparison, ECPT achieves only a 25% reduction in page walk cycles when using 4KB pages and 20% reduction when using THP. In other words, *LVM outperforms ECPT in terms of page walk cycles by 2×*. Overall, LVM achieves significant MMU overhead reduction over Radix and ECPT.

Page Walk Traffic. Next, we examine the MMU overhead reduction achieved by LVM through an analysis of page walk traffic. Figure 3.11 presents the number of memory requests sent to the cache hierarchy due to page walks. Results for 4KB configurations are normalized to Radix with 4KB pages, and THP configurations are normalized to Radix with THP. LVM drastically reduces memory traffic associated with page walks. Compared to Radix with 4KB pages, LVM reduces page walk traffic by an average of 43%. For THP, the reduction is similarly significant at 34%. These reductions represent substantial improvements in MMU efficiency.

When comparing LVM to the state-of-the-art hashed page table, ECPT, the benefits are even more pronounced. ECPT, on average, generates $1.7\times$ and $2.1\times$ more memory accesses for page walks compared to Radix with 4KB and THP configurations, respectively. This is due to ECPT’s design, which employs multiple parallel requests, trading off reduced latency for significantly increased bandwidth usage. In contrast, LVM not only surpasses ECPT in terms of end-to-end performance but also achieves a substantial reduction in MMU traffic, with $2.9\times$ and $3.1\times$ fewer memory accesses compared to ECPT 4KB and THP configurations, respectively.

Finally, while not depicted in Figure 3.11, LVM requires only 1% additional page walk-induced memory traffic compared to the ideal page table design, underscoring its efficiency.

Cache Performance. To quantify the impact of page walk traffic, we examine the misses per kilo instructions (MPKI) for L2 and L3 caches shown in Figure 3.12. The results demonstrate that LVM achieves high cache efficiency. Specifically, the L2 and L3 MPKI of

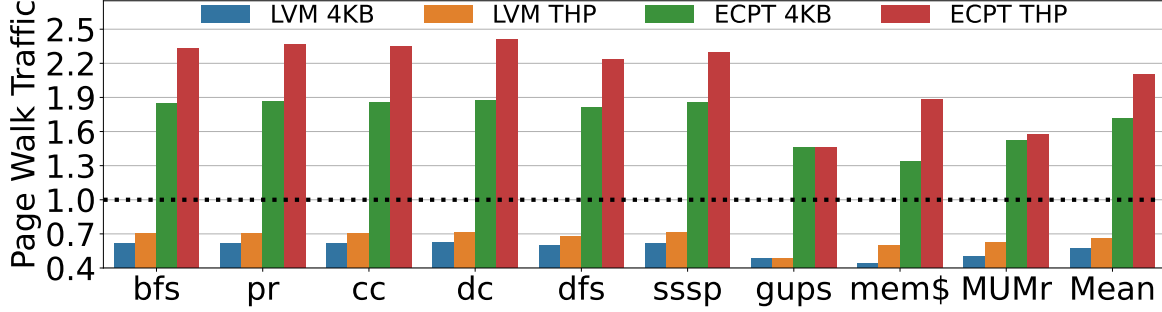


Figure 3.11: Page walk traffic relative to radix. Results are normalized separately to Radix 4KB and THP.

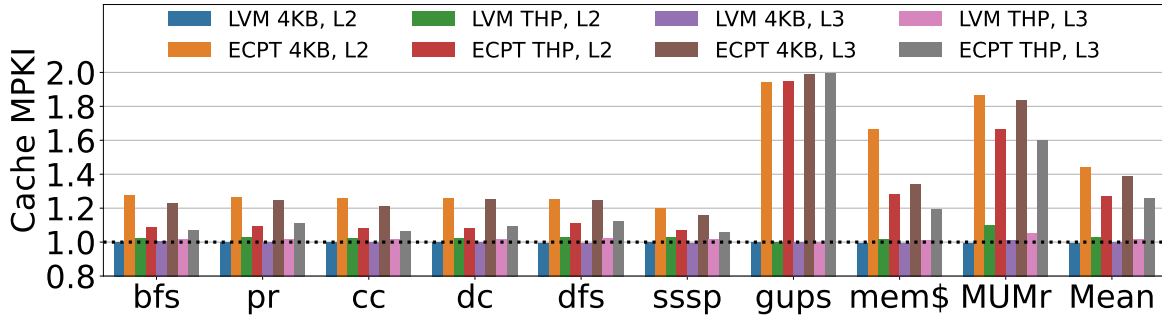


Figure 3.12: Cache MPKI relative to radix page tables.

LVM is, on average, within 1% of that of radix. In contrast, ECPT exhibits significantly higher MPKI, with increases of 44% in L2 and 40% in L3. This behavior stems from ECPT’s reliance on parallel accesses to locate a translation, resulting in additional page table entries being fetched into the caches. This effect is particularly pronounced in workloads with large page table entry working sets, such as GUPS, memcached, and MUMmer, where ECPT leads to on average 72% and 59% higher MPKI in L2 and L3, respectively. This is expected: as workload memory footprints grow, ECPT’s larger PTE working set exceeds the capacity of the caches, causing cache pollution and contention between data and PTEs. In contrast, LVM efficiently locates translations with a single page table entry lookup in the common case, thereby ensuring low translation latency while maintaining high cache efficiency.

TLB, PWC and LWC Miss Rates. In the simulation, the L2 TLB miss rates (57.5%–99.4%) remain nearly identical across different configurations (Radix, ECPT, and LVM) since they solely affect the page walk process. Radix PWC suffers medium to high miss rates (59.7%–99.6%) at the PMD level due to limited coverage, while the upper levels enjoy high hit rates (96.0%–99.9%) across all workloads. Among the workloads, graphs exhibit moderate TLB and PWC miss rates of about 60% to 70%, whereas MUMmer, memcached and GUPS exhibit very high TLB and PWC miss rates of over 90%. Instead, the LWC of

LVM enjoys high hit rates of above 99% across all applications due to the small size of the learned index.

Connecting PTW to L1/L2 cache. We consider an additional configuration that connects the MMU page table walkers to the L1 cache instead of the L2 cache. On average, LVM outperforms radix by 11% by connecting to the L1 cache and by 14% when connecting to the L2. Connecting to the L1 enables faster page walks due to L1 cache hits for page table entries but it significantly increases cache pressure since page table entries contend at the L1 cache with regular data. Specifically, while the MMU overhead reduces in general, radix benefits a bit more as its sequential multi-access page walks can now potentially hit at the L1 cache, which has lower latency. However, the page walk traffic at the L1 drastically increases the L1 MPKI by 59% for radix, while LVM performs better, increasing L1 MPKI less, by 38%. This is because LVM sends close to 43% less page walk traffic to the caches as we discussed earlier (Section 3.7.1). Overall, LVM significantly outperforms radix irrespective of configuration.

3.7.3 Learned Page Table Characterization

Collision Rates. Collisions during page walk occur when multiple VPNs are predicted by the learned index to reside in the same location in the page table. To characterize the effectiveness of LVM, we measure collision rates and compare LVM to a hash table that has a load factor of 0.6 and uses the state-of-the-art hash function Blake2 [19]. On average, LVM achieves a minimal collision rate of 0.2% for 4KB and 0.6% for THP. This is a drastic improvement over regular hash tables, which have a collision rate of 22% for 4KB and 19% for THP. Notably, several graph workloads enjoy near-zero collision rates under LVM. Overall, LVM achieves a very low collision rate by tailoring to applications’ virtual address spaces.

Collision Resolution. When there is a collision during address translation, LVM searches for the actual location of the translation entry corresponding to the VPN. As we discussed in Section 3.4, LVM bounds the number of memory accesses involved in such a search by enforcing an upper bound of $C_{\text{err}} = 3$ when building the model. To understand how this upper bound behaves in practice, we characterize the number of additional memory accesses across our workloads. We identify that the average number of additional memory accesses per collision across all workloads is 2.36 memory accesses. Since LVM has collision rates of only 0.6%, it requires a single-access 99.4% of the time on average across all workloads.

Index Size	bfs	pr	cc	dc	dfs	sssp	gups	mem\$	MUMr
LVM 4KB	112	112	112	112	112	112	96	112	128
LVM THP	192	192	192	192	112	192	112	112	160

Table 3.2: Index size of LVM in bytes.

Index Size, Cacheability, and Scaling. In Table 3.2, we present the index size in bytes during the steady state. The results show that LVM requires a minimal index size of 112 bytes and 162 bytes on average for 4KB pages and THP, respectively. To determine

the appropriate size for the LVM page walk cache (LWC), we analyze the steady-state and peak index sizes. The peak index size during the initial training phase averages 570 bytes, after which it stabilizes to the steady-state size. Because the peak and steady-state indexes are small, the LWC can be sized to comfortably accommodate the entire index. As discussed in the next section, the LWC required by LVM is $3\times$ smaller than that of radix page tables.

Importantly, the learned index size is not dependent on the memory footprint. For example, memcached uses six times more memory than MUMmer but requires a smaller index. To better showcase this property, we conducted a scaling study based on memcached, where the working set was scaled from 32 GB, to 64 GB, 128 GB, and 240 GB. The corresponding steady-state index sizes were all 112 bytes. This contrasts sharply with radix page tables, which require linearly more entries in the page walk caches as the memory footprint grows, making LVM a future-proof solution.

LVM Overheads in the OS. The operating system is responsible for managing LVM for each application throughout its lifetime. The computational costs are included in the simulation as they occur, including the initialization of the learned index and ongoing management during simulation. For 4KB pages, the management overhead averages 1.17% of the total execution time across all workloads, with a peak of 1.91% for dfs. A primary reason for the low computational cost of LVM is its minimal and lightweight retraining, and beyond the short window of simulation, the overhead will be even more diluted. For THP, the percentage of time spent on management is less than 0.01% for all workloads since the OS has significantly fewer pages to manage. Overall, the management overhead of LVM is negligible.

We further use our OS prototype to run applications end-to-end for their complete runtime, far beyond what is possible in simulation, and measure the frequency and the computational cost of retrainings. Across workloads, the frequency of retraining is very low. Retrains occur at most 3 times, and 2 times on average. Importantly, retrains are very fast, less than 1.7ms on average and with the worst case of 1.9ms. This is because application address spaces remain largely unchanged after initialization, with expansions handled efficiently without retraining (Section 3.4.3). We further perform a tail latency study with memcached. Our results show that LVM computational costs do not affect even the 99th percentile tail latency.

Memory Consumption. Beyond the space of the page tables entries, every page table structure has additional memory overhead—radix page tables require space for upper-level page tables, and hashed page tables require over-provisioning of the hash tables to sit below an occupancy threshold. We compute the overhead of memory usage of LVM by calculating the additional space used in comparison to the absolute minimum required: eight bytes for every translation entry that maps a physical page. The overhead of LVM comes from the gapped array organization, which, in the worst case, uses an additional 1.3 times the space for PTEs than what is absolutely necessary. For example, LVM uses at most an additional 12MB of memory compared to the minimum required page table for MUMmer with 20GB of memory footprint, while ECPT uses an additional 27MB. Note that, as we discussed in Section 3.4.2, gapped page tables are not required to be physically contiguous.

Memory Fragmentation As we discussed in Section 3.4, LVM adapts to the available physical memory contiguity in the system. To this end, we evaluate a range of memory fragmentation levels. We first limit the available physical contiguity for LVM allocations to at most 256 KB, which represents 30% of free memory even in highly fragmented production data centers as shown in Figure 3.3. Furthermore, we study three even more severe levels of fragmentation with a free memory fragmentation index (FMFI) [69, 117] of 0.8, 0.85, and even 0.9. When physical contiguity is limited, LVM dynamically adjusts the number of leaf nodes based on the available contiguity during allocation. Importantly, per-node coverage of LVM remains high, resulting in high LWC hit rates (above 99%). In practice, such severe fragmentation is not maintainable as the OS immediately compacts memory to replenish multi-KB contiguity. Overall, LVM’s performance remains the same in production environments and even at more aggressive fragmentation levels due to its ability to dynamically allocate small non-contiguous gapped page tables based on the available physical contiguity.

3.7.4 Hardware Characterization

We implemented the page walker of LVM in RTL and synthesized it in a commercial 22nm PDK, and used CACTI [22] to generate the SRAMs of the caching structures. Our results show that a page walk model calculation and lookup in the LWC can be completed in 2 cycles at 2GHz and a single LVM page walker requires $0.000637mm^2$ of area. The LWC of LVM requires $0.00364mm^2$ of area and $0.588mW$ of leakage power, both lower than Radix PWC. The hardware structures of LVM achieve $3.0\times$, $1.5\times$ and $1.9\times$ improvement over radix in terms of size in bytes, area, and power, respectively.

3.7.5 Additional Comparison to Prior Work

Beyond ECPT, we further compare LVM to additional prior work.

Comparison to ASAP

ASAP [142], which relies on both virtual and physical contiguity to allocate leaf page tables in contiguous physical memory and to prefetch them during page walks. ASAP performs worse than both ECPT and LVM, with average slowdowns of 3% and 8%, respectively. This performance degradation is primarily due to the prefetcher introducing additional memory traffic on top of the standard page walk. Moreover, ASAP requires the operating system to allocate large contiguous regions of physical memory—potentially hundreds of megabytes—at VMA creation time. In practice, such large contiguous allocations are difficult to guarantee (Chapter 2).

Comparison to Midgard

Midgard [76] reduces address translation overheads by creating an intermediate Midgard address space for data in the cache hierarchy, thus avoiding page walks for cache hits.

However, Midgard still relies on radix page tables to access main memory. We evaluated an ideal Midgard that does not include any OS overheads. Midgard achieves, on average, a modest speedup of 3% compared to Radix, and LVM outperforms Midgard by 11%. This is because memory-intensive workloads exhibit a high L3 MPKI, which requires frequent radix page table walks. To further test the scalability of Midgard with larger caches, we further evaluated a Midgard design with twice the LLC slice size per core, resulting in 4MB LLC slices. Both LVM and Midgard achieve higher speedups. However, even with such a large LLC capacity, Midgard achieves only modest speedups over radix of 5%, and LVM still outperforms Midgard by 11%. Overall, we believe that Midgard is orthogonal to LVM as a potential design can rely on Midgard combined with LVM page tables for maximum benefits.

Comparison to Flattened Page Tables

Flattened Page Tables [158] (FPT) reduce the number of memory accesses per page walk by folding adjacent levels of Radix Page Tables, but requires 2MB or 1GB pages for folding to succeed. As we show in Figure 3.3, there is a limited supply of 2MB or larger physical contiguity in data centers, which makes flattening difficult in practice. Furthermore, since it is unlikely that the high latency caused by memory reclaim or compaction can be tolerated at page fault time, the success rate of 2MB or larger physical memory allocations will be even lower. Instead, LVM does not require such large contiguity. Another drawback of FPT is that it constrains the concurrent use of multiple page sizes, e.g., 4KB, 2MB, and 1GB in the same virtual region, depending on which levels of the radix page tables are folded. Real-world applications benefit from multiple page sizes (Chapter 2).

To demonstrate the difficulty of FPT allocations, we ran a simulation of FPT where both L4+L3 and L2+L1 radix page tables are flattened using 2MB pages. Our results show that in lightly fragmented environments, LVM outperforms FPT by 5% while in highly fragmented environments, the performance of FPT degrades to that of radix. The main challenge of FPT is that 2MB page table allocations compete with 2MB page allocations for application data, which, as a result, can lead to FPT allocating regular radix page tables, especially as fragmentation increases. On the other hand, LVM is responsive to the available memory contiguity (Section 3.4.2), resulting in single-access page walks even with limited physical memory contiguity. Finally, LVM flexibly supports all page sizes.

3.8 Related Work

A large body of work has focused on reducing the computational cost of address translation [25, 63, 65, 68, 74, 76, 79, 85, 107, 117, 136, 137, 141, 155, 156, 158, 160, 161, 162, 168, 170, 187, 193, 194, 195, 199, 209, 210, 225, 234] (Chapter 2) through alternative page table structures, large and coalesced translation entries, and huge page support in the OS.

Relevant to this work is [136], which leverages machine learning techniques to improve the allocation of huge pages. Other works have focused on the application of learned indexes for storage FTL [195] and persistent memory [49, 131]. The most relevant prior work to

LVM is a short paper [141] that applied a learned index [113] using neural networks to predict translation locations in radix page tables. This approach did not support insertions and fell back to traditional page table walks on mispredictions, effectively functioning as a prefetching mechanism. The paper concluded that existing learned indexes were not suitable for address translation due to their large model sizes and high computational requirements resulting in 120 cycles to produce a PTE location. While the idea was forward-looking, we believe the primary pitfall of that work was its adherence to radix page tables for backward compatibility with existing operating systems. It further relied on an off-the-shelf learned index with heavyweight neural network functions originally designed for databases, which introduced significant implementation challenges. In contrast, LVM reimagines the end-to-end page table structure to unlock a highly efficient and practical design.

3.9 Discussion and Future Work

Learned indexes are a promising new class of data structures that accelerate indexing operations in software. LVM pioneers the first practical learned index for hardware, specifically for address translation. To make learned indexes suitable for fast hardware lookups, LVM employs cost models that bound index depth and size, introduces efficient mechanisms for key updates, fast lookups, and relaxes the need for contiguous memory allocations. These advances are essential to applying learned indexes in hardware.

Looking ahead, the LVM framework can also benefit other hardware caching structures indexed by virtual addresses, such as TLBs, branch target buffers (BTBs), and L1 caches. Beyond this, we envision extending LVM to physically indexed structures like last-level caches. Such structures often suffer from hash-table-like collisions that cause conflict misses and reduce hit rates. By leveraging lightweight machine learning, the LVM framework offers a promising direction to mitigate these collisions and improve performance. We leave this exploration to future work.

3.10 Conclusion

We presented Learned Virtual Memory (LVM), a page table structure that effectively provides optimal single-access address translation. LVM relies on a novel learned index model that tailors address translation to the virtual address space of applications. Our evaluation shows that LVM performs within 1% of an ideal page table.

Chapter 4

Pensieve: Towards Multi-Tier Memory Systems

4.1 Introduction

DRAM is a major contributor to data center server costs. In the early 2020s, DRAM comprised 40–50% of server costs [150, 216]. Recently, this proportion has surged as DRAM price has doubled [43, 62, 139, 208], with further price increases anticipated. Driven by the immense demands of emerging AI applications, DRAM is also increasingly scarce as demand outstrips supply. Therefore, data center operators are urgently searching for strategies to reduce their DRAM footprint.

To reduce DRAM footprint, hyperscalers increasingly rely on *tiered memory*, demoting infrequently accessed or latency-tolerant data to slower, cheaper tiers. Hyperscalers use tiered memory to reduce expensive DRAM usage so that more jobs can be scheduled on fewer resources, while ensuring that application performance does not degrade below a predetermined service-level objective (SLO). To date, two-tier fast/slow memory architectures have dominated research in both academia and industry [2, 37, 55, 83, 109, 121, 123, 128, 145, 171, 173, 184, 196, 207, 212, 223, 226, 230, 238, 240]. They include both traditional swap-based tiers like SSDs or compressed memory (e.g., zswap [126]) and emerging, directly-addressable technologies like Compute Express Link (CXL) paired with recycled previous-generation DRAM [108, 213, 238].

Two-tier approaches, however, are fundamentally limited in the amount of DRAM they can save. SSD swap-based tiers impose long access latencies and expensive page faults when accessing offloaded data. This creates a "performance wall" on the amount of data they can offload without severely degrading application performance. Additionally, the SSD's finite endurance budget may constrain the rate at which data can be swapped out, limiting DRAM savings. CXL-attached memory offers superior performance, but its offload potential is limited by its capacity, dictated by both physical form factors and limited supply. As a result, two-tier systems using SSDs or CXL alone are unable to offload sufficient data to meet hyperscaler needs in the current DRAM crisis.

A three-tier system combining DRAM, CXL, and SSD can overcome the above limita-

tions. Compared to a DRAM+CXL configuration, adding an SSD tier allows the coldest data to be offloaded to SSD, freeing space in CXL for more data to be offloaded from DRAM. Compared to a DRAM+SSD configuration, the added CXL tier effectively serves as a performant buffer, overcoming the SSD performance wall. This architecture thus promises a pathway to achieve greater DRAM savings than a two-tier configuration.

Adding a third tier, however, is not merely an extension of the two-tier problem. Designing a performant and robust three-tier system is inherently harder, not just because there are more decisions to make (e.g., the added dimension of *where* to migrate to, in addition to *when* and *how much* to migrate), but more importantly, because of the tight coupling between these decisions. Consequently, any individual decision—even one that seems entirely reasonable in isolation—can have unexpected consequences and push the system in an unintended direction. For example, a decision to slow down CXL $\rightarrow$ SSD demotion to alleviate the performance hit of page faults without careful orchestration quickly exhausts CXL capacity. This might force more aggressive CXL $\rightarrow$ DRAM promotion, pushing CXL bandwidth usage into saturation and causing page faults to become even more expensive (as reading pages into CXL consumes that saturated bandwidth). Therefore, a successful design must reduce this complexity and manage the coupling.

We present **Pensieve**, to our knowledge the first three-tier memory system designed for data centers. Pensieve employs three key design principles crafted to combat the complexity of a three-tier system and manage coupling in a principled way. As a result, we arrive at a system that is easy to reason about, can be precisely controlled, and successfully achieves all of our operational requirements. Most importantly, Pensieve delivers significantly greater DRAM savings than either a DRAM+CXL or a DRAM+SSD two-tier system.

First, Pensieve enforces a **strict data hierarchy** by disabling direct page migration between the top and bottom tiers (i.e., DRAM $\leftrightarrow$ SSD) in the common case. This significantly prunes the decision space by entirely removing the *where-to* question from any promotion or demotion path. Furthermore, demoting data through the CXL tier acts as a filter against costly placement errors. It prevents pages not yet deemed to be hot from wasting precious DRAM space, and protects against premature DRAM $\rightarrow$ SSD demotions that could trigger expensive page faults. Ultimately, this pruning not only reduces system complexity but also ensures the decisions the system makes are inherently low-risk. This hierarchy also enforces an approximate “temperature gradient” across the system: DRAM for hot data, CXL for warm data, and SSD for cold data, which makes the system easier to reason about and monitor.

Second, we decompose the three-tier hierarchy into **two loosely coupled subsystems**. The DRAM $\leftrightarrow$ CXL subsystem aims to aggressively save DRAM by proactively demoting as many pages as possible from DRAM to the CXL tier. Meanwhile, the CXL $\leftrightarrow$ SSD subsystem aims to maintain high CXL utilization so that expensive page faults and SSD write traffic are minimized. This separation reduces the complexity of the global optimization problem across three tiers into two distinct tasks with simplified goals and constraints.

Finally, Pensieve governs the entire architecture through a **single dynamic control knob**: the aggressiveness of DRAM $\rightarrow$ CXL demotion. By locking the opposing CXL $\rightarrow$ DRAM promotion path to a static threshold, we eliminate “knob contention” and prevent

the performance oscillations (or page "ping-ponging") that occur when two dynamic knobs chase each other. Furthermore, the single knob provides *indirect* control over the bottom tier. Because CXL acts as a shared pressure valve, regulating the "inflow" of demoted data from DRAM naturally dictates the capacity-driven "outflow" of evicted data to the SSD. Ultimately, this reduces a multi-variable optimization problem into a single tuning parameter, ensuring fast and stable system convergence.

By combining these three design principles, Pensieve effectively manages a three-tier memory system in a controlled manner, resolving the challenges of complexity and coupling. When evaluated against a set of memory-intensive workloads (including production workloads and one derived from hyperscaler production memory access patterns [185]), Pensieve achieves significantly greater DRAM savings. On average, it offloads 33% of the application memory footprint from DRAM, substantially outperforming standalone DRAM+CXL (21%) or DRAM+SSD (18%) two-tier setups. Pensieve accomplishes this while keeping application performance degradation within an acceptable range (≤5%) and respecting SSD endurance constraints. Moreover, Pensieve is highly robust, maintaining stable operations even under dynamic workload shifts and environmental changes.

4.2 Background

In this section we provide background on traditional two-tier memory systems and explain why they are inherently limited in their ability to offload data from DRAM.

4.2.1 Swap-Based Two-Tier Systems

Many hyperscalers have adopted swap-based two-tier memory architectures, which migrate data from DRAM to a non-byte-addressable tier, such as SSDs [216] or compressed memory [119]. Because the CPU cannot directly access data in these tiers, any access triggers a *page fault* to bring the entire page back into DRAM. Swap-based tiers often have long access latencies—for example, even high-end SSD latencies average $\sim 100 \mu\text{s}$. Both the page fault and hardware latency, coupled with the requirement that the entire page (rather than just the requested cacheline) must be migrated, add orders of magnitude to the total access penalty. As a result, the volume of data that can be offloaded without compromising application performance is limited.

Furthermore, SSDs have a finite number of Program/Erase (P/E) cycles; exceeding this limit leads to device wearout. To avoid incurring operational and financial costs to replace server-local SSDs, data center operators aim to align SSD operational lifetime with that of the server. SSD writes during this operational lifetime must fit within the endurance budget, which is calculated based on the SSD lifetime, its P/E cycle rating and the allocated swap partition. Memory tiering systems must consider the endurance budget (§4.3.1).

4.2.2 Byte-Addressable Two-Tier Systems

Byte-addressable tiers like CXL-attached memory, on the other hand, maintain DRAM-like memory semantics and remain CPU-accessible at byte-granularity [46, 51], while providing access latency that is only $2 - 3\times$ (instead of orders of magnitude) higher than DRAM. As a result, they do not suffer from the same fundamental performance wall as swap devices when offloading data from DRAM. However, CXL-attached memory—specifically previous-generation DDR4 (a common choice for high-performance DRAM offloading [134, 213, 239] (Chapter 5))—is heavily constrained by its capacity, which restricts potential DRAM savings.

This capacity limitation stems from two factors. First, physical form factors restrict the number of CXL-attached memory DIMMs to a specific value. This constraint, coupled with the smaller capacities of older-generation DIMMs, caps the CXL-attached DDR4 memory capacity that can be populated on a given machine. Furthermore, as modern servers support larger and more main DRAM DIMMs, this results in a relatively small CXL-to-DRAM capacity ratio. For example, the OCP Hyperscale CXL Memory Expander provides a maximum of 384 GiB of CXL-attached DDR4, with 32 GiB DIMMs and up to 12 channels [39]. Conversely, an Intel Granite Rapids-AP server supports 1.5 TiB of DDR5 main memory [217] using 128 GiB DIMMs (and can go even higher with 256 GiB DIMMs). Thus, for a server with a single CXL controller, the CXL memory capacity can only go up to 20% of total memory capacity.

Second, previous-generation memory becomes increasingly scarce as it goes out of production. Whether purchasing these modules from vendors or repurposing used ones after they retire, deployments at scale are strictly constrained by the supply of no-longer-in-production DDR4 modules.

Ultimately, this limited CXL capacity dictates the maximum possible DRAM savings in any byte-addressable two-tier system constructed from older-generation memory. Guided by these hardware and supply-chain realities, we assume a 25% CXL to 75% DRAM capacity ratio throughout this work, representing a reasonable setup for our server environment.

4.3 Requirements and Challenges

A three-tier memory system layering CXL-attached memory and SSDs below main DRAM can overcome the limitations of swap-based and byte-addressable two-tier systems. Compared to a swap-based system, a three-tier system can use CXL memory as a highly performant buffer to overcome the page fault performance wall. Compared to a byte-addressable two-tier system, it can leverage SSD as additional space to offload the coldest data, overcoming the capacity constraints of CXL memory. It thus promises a pathway to greater DRAM savings while limiting performance impact.

To the best of our knowledge, no system has combined these tiers in a single architecture to maximize DRAM efficiency, especially in a data center setting. This gap exists because designing a well-behaved, robust three-tier system involves non-trivial challenges. In this section, we first define the three-tier memory problem based on our specific goals and

requirements (§4.3.1). We then identify the challenges of solving this problem, which stem from the inherent complexity and tight coupling of a three-tier system (§4.3.2).

4.3.1 Problem Formulation

Pensieve targets a three-tier memory system in a large-scale data center environment, where each machine services a diverse set of jobs assigned by a cluster-level scheduler. We assume each machine has a predetermined amount of DRAM, CXL-attached memory, and local SSD storage. Notably, the CXL tier is capacity-constrained, representing 25% of the byte-addressable memory capacity (see §4.2.2) with the rest provided by DRAM.

Our goal is to design a system that uses memory tiering to execute a set of workloads with the minimum amount of DRAM possible – i.e., *maximizing the DRAM savings* – while ensuring that system performance does not degrade below pre-defined SLOs. Maximizing DRAM savings increases a machine’s “schedulable memory capacity”, allowing more jobs to be scheduled on it, thus improving end-to-end infrastructure efficiency. As the cost per GB of DRAM is significantly higher than both SSDs and decommissioned DDR4, DRAM savings are directly linked to monetary savings.

This problem can be formulated as a constrained optimization problem:

Objective: Maximize the total DRAM savings $\mathcal{D}$.

Constraints:

1. **Performance Bound:** For each metric p_i in the set of monitored performance metrics $\mathcal{P}$, the observed value must remain within its respective pre-defined threshold T_i : $\forall p_i \in \mathcal{P}, \quad p_i \leq T_i$
2. **SSD Endurance:** The *average* SSD write traffic $\mathcal{W}$ from swapping must not exceed the threshold determined from the SSD write endurance budget $\mathcal{E}$.
3. **CXL Capacity:** The data placed in the CXL tier must always remain below the effective capacity $\mathcal{C} \times (1 - \alpha)$, where α represents the safety buffer.

Because the system must support diverse workloads, each measuring its performance with different metrics, we use a set of system-level **indicator metrics** to estimate the slowdowns caused by tiering instead of relying on end-to-end application performance. By constraining the system so that these indicators remain within their bounds, we indirectly ensure that application performance remains within SLOs.

Another critical constraint concerns the SSD endurance budget (§4.2.1). We must limit the *average* SSD write rate induced by tiering traffic to avoid premature device failure. While instantaneous write traffic may occasionally spike above this threshold, the system must ensure that the long-term average remains within budget.

Finally, we must respect the physical capacity of each memory tier. Our objective of maximizing DRAM savings inherently pushes DRAM usage below the physical limit. Furthermore, modern SSDs are significantly larger than local DRAM, providing ample space for swapped data. Consequently, only the CXL tier demands a capacity constraint. Our system must ensure that CXL occupancy remains within its physical capacity, minus a safety buffer to maintain stability and accommodate spikes of page migrations.

4.3.2 Challenges

Constructing a three-tier memory system is not a linear expansion of the two-tier problem; rather, adding a third layer makes the problem inherently harder. On a surface level, there are more decisions to make. Promotion and demotion logic must now account for multiple tier combinations (e.g., DRAM $\leftrightarrow$ CXL, CXL $\leftrightarrow$ SSD, and DRAM $\leftrightarrow$ SSD). In addition to *when* to demote or promote a page, there is the added dimension of *where* to move it. For instance, a page demoted from DRAM can be routed to either CXL or SSD, and a newly allocated page can be placed in either DRAM or CXL.

However, the bigger challenge lies in the fact that these decisions are **tightly coupled**. A critical requirement for any robust system is the ability to control it and steer it back into its “operational range” when things go wrong. However, in a three-tier system an individual decision—even one that seems entirely reasonable in isolation—can have unexpected consequences and push the system in an unintended direction. This tight coupling makes designing a stable and robust system difficult. Consider the following example scenarios:

Bandwidth Paradox: The CXL bandwidth usage is too high, so the system slows down CXL $\rightarrow$ DRAM promotion to conserve it. However, because pages aren’t leaving CXL, the CXL tier quickly fills up. This forces CXL $\rightarrow$ SSD evictions and directs more application memory access to the CXL tier, and the combination consumes even more CXL bandwidth. The attempt to correct the system pushed it further out of its acceptable range.

Utilization Trap: The CXL utilization is low, so the system directs more new allocations to the CXL tier instead of DRAM to drive utilization up. However, because CXL is slower, performance drops. To reduce this performance impact, it starts promoting more pages from CXL back to DRAM. In the end, it hasn’t meaningfully increased CXL utilization, but has caused excessive data movement.

Performance Spiral: Performance is suffering due to too many page faults, so the system slows down CXL $\rightarrow$ SSD demotions, which quickly exhausts CXL capacity. The system responds by increasing the aggressiveness of CXL $\rightarrow$ DRAM promotions, saturating CXL bandwidth. Page faults slow down even more, as they contend for already saturated CXL bandwidth, further exacerbating the performance problem.

As these examples demonstrate, we need a principled way to reduce this complexity and manage the coupling. Doing so will allow us to steer the system toward our intended targets (e.g., lower bandwidth usage, higher CXL utilization, or lower performance degradation) without having to manually account for all the permutations of coupled behaviors.

Failures of Naive Solutions

One might be tempted to borrow solutions from existing multi-tier systems. For example, the computer architecture community has extensively studied the management of multi-level CPU caches [82]. However, hardware caches operate with a fundamentally different goal: they try to maximize upper-level cache occupancy, only evicting data to make room for an incoming item. In contrast, our system strives to proactively create as much space as possible in the “top-level cache” (DRAM), while taking performance, SSD endurance,

and other factors into account. The proactive migration creates a totally different problem with more degrees of freedom and complexity.

A naive three-tier setup, cobbled together using Linux’s default mechanisms (Transparent Page Placement (TPP) [145] for DRAM-CXL management and swap for the CXL-SSD path) exemplifies this limitation. When evaluated against a *memcached* workload (setup described in §4.6), the system failed to reduce DRAM usage meaningfully. The DRAM tier remained over 99% full, while SSD utilization was negligible (less than 0.3% of the application footprint). This occurs because TPP maximizes DRAM occupancy like a traditional cache, and Linux treats swapping as a reactive “failure path” triggered only when upper tiers are exhausted. Ultimately, managing three-tier memory as a conventional cache hierarchy fails to achieve our goal.

Another straightforward approach is to layer two independent two-tier management systems. We evaluated this design by combining two separate proactive demotion control loops, DRAM $\rightarrow$ CXL and CXL $\rightarrow$ SSD (configured identically to Pensieve).¹ By employing proactive demotion, this setup overcomes the shortcomings of TPP and achieves some DRAM saving (23%). However, it is not as effective as Pensieve (38%). This outcome is unsurprising: because the CXL $\rightarrow$ SSD loop proactively demotes pages from CXL without coordinating with the DRAM $\rightarrow$ CXL loop, it causes under-utilization of the higher-performance CXL tier (33%, compared to 80% of Pensieve), which hurts application performance and creates unnecessary SSD writes, aggravating endurance concerns. This further highlights the importance of explicitly managing the coupling and interactions between different policy decisions within a three-tier architecture.

4.4 Design

We design our three-tier memory system by systematically reducing this complexity and managing coupling in a controlled manner. We first give an overview of Pensieve’s operation (§4.4.1) and discuss its key high-level design decisions (§4.4.2). We then present specific aspects of the design, including the choice of indicator metrics (§4.4.3), the control algorithm (§4.4.4), and the configuration of different data migration paths (§4.4.5).

4.4.1 System Overview

Pensieve, illustrated in Figure 4.1, makes three high-level design decisions to reduce decision complexity and manage coupling: a hierarchical data path, decoupled DRAM $\rightarrow$ CXL and CXL $\rightarrow$ SSD management, and a single dynamic control knob for the whole system. The hierarchical data path (shown by the black arrows in the data path box) means that new pages are allocated in the DRAM tier; as they cool down, Pensieve demotes them first to the CXL tier and eventually to the SSD tier. Conversely, as pages are accessed more intensely, they are promoted from SSD to CXL, and eventually back to DRAM. The direct DRAM-SSD path is disabled in the common case.

¹To achieve this, we had to fix several broken Linux mechanisms regarding three-tier support. We discuss these implementation details in §4.5.1.

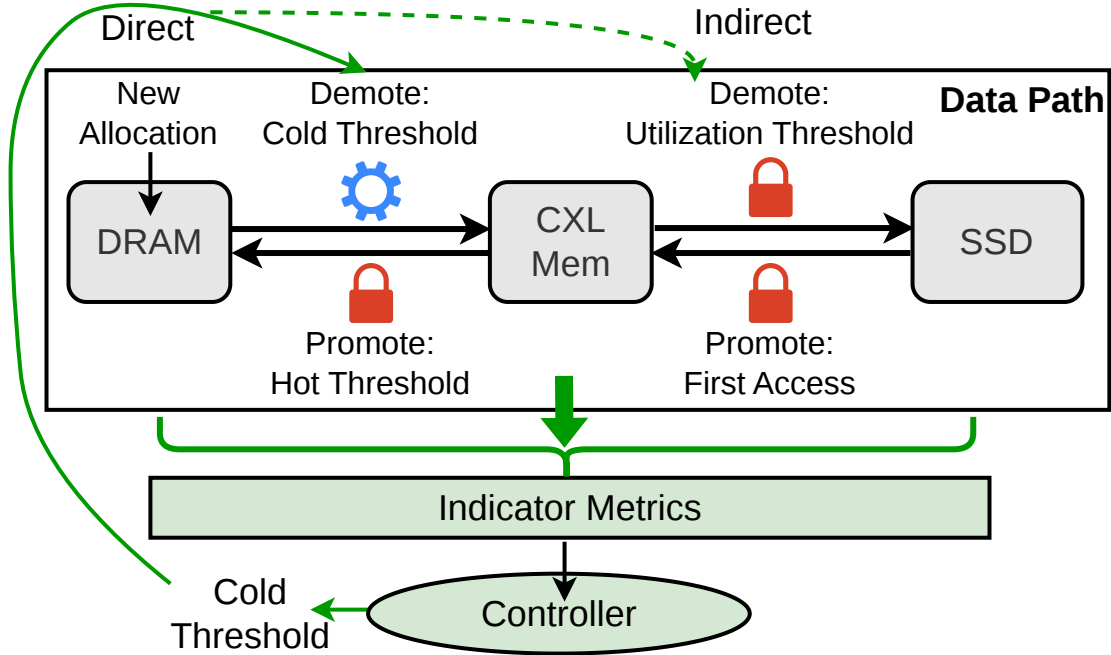


Figure 4.1: Pensieve high-level design. Black arrows mark data path operations. The control paths are shown in green.

The system decouples the operation of the DRAM $\leftrightarrow$ CXL and CXL $\leftrightarrow$ SSD subsystems, allowing them to be separately optimized for different goals: DRAM $\leftrightarrow$ CXL aims to minimize the DRAM footprint by proactively demoting data based on a coldness threshold. CXL $\leftrightarrow$ SSD aims to keep CXL memory utilization high by enforcing utilization thresholds, providing performance and endurance benefits. DRAM $\rightarrow$ CXL demotion is tuned dynamically (blue gear), whereas the other three data paths are managed statically (red locks).

The overall hierarchy is controlled using a single control knob, the coldness threshold for DRAM $\rightarrow$ CXL demotion. The system monitors a set of indicator metrics, which the controller uses to dynamically adjust the coldness threshold, thus directly influencing DRAM $\rightarrow$ CXL operations and indirectly influencing CXL $\rightarrow$ SSD operations. This carefully managed coupling ensures that the system is within its operational range while achieving our design goals.

4.4.2 High-Level Design Decisions

Pensieve’s high-level design choices serve to prune the design space and simplify the system architecture while ensuring (through data-driven analysis) that all performance and operational requirements are met.

Hierarchical Data Migration Path

Pensieve enforces a strict hierarchy by disabling direct page migration between the top and bottom tiers (i.e., DRAM $\rightarrow$ SSD and SSD $\rightarrow$ DRAM) in the common case.² This choice significantly prunes the large decision space: during demotion or promotion, Pensieve no longer needs to decide *where* to migrate (e.g., whether to demote a DRAM page to CXL or directly to SSD)—only *when* and *how much* to move. This approach offers significant performance and operational benefits:

CXL tier as a filter. The CXL tier acts as a “cooling-off” zone, ensuring only genuinely cold pages are demoted to SSD to avoid order-of-magnitude-slower page faults for warm data. Similarly, promoting a page directly to DRAM upon a page fault—before confirming it is truly hot—can waste precious DRAM capacity. On a randomly chosen production server in our data center, we observed that 34% to 57% of swapped-in pages do not see subsequent accesses after two minutes. The CXL tier again serves as a buffer, allowing the system to verify a page’s hotness before granting it a spot in DRAM. Such hierarchical simplification removes choices in a way that ensures the placement decisions the system makes are inherently low-risk.

Temperature gradient and observability. This hierarchy enforces an approximate “temperature gradient”: DRAM for hot data, CXL for warm data, and SSD for cold data. This makes the system much easier to reason about and monitor.

To further enforce this hierarchy, we also restrict new page allocations to the DRAM tier, as shown in Figure 4.1. Pages are only placed in lower tiers once they are identified as cold. This decision is driven by the recognition that newly allocated pages are almost universally hot, as applications generally allocate memory with the intent of immediate use. Interestingly, even pages that appear cold immediately following allocation often become hot suddenly. These pages often back the free lists of user-space allocators (e.g., *tcmalloc*); consequently, they may be used to house hot objects at any moment. Since there are no reliable signals to distinguish which new pages will remain cold, DRAM is the most robust initial placement.

Decoupled DRAM $\leftrightarrow$ CXL and CXL $\leftrightarrow$ SSD Management

To further simplify system management, we decompose the three-tier hierarchy into two loosely coupled subsystems. They are optimized for different goals and employ distinct logic. The **DRAM $\rightarrow$ CXL subsystem** aims to minimize the DRAM footprint by aggressively and proactively demoting as many pages as possible from DRAM to the CXL tier, provided the system remains within its constraints.³ Its operation is governed by a dynamically tuned page coldness threshold: any pages colder than the threshold are proactively demoted. This effectively achieves the goal of maximizing DRAM savings. The **CXL $\rightarrow$ SSD subsystem** aims to maintain high CXL utilization through static

²Under near-out-of-memory (OOM) conditions, we allow SSD $\rightarrow$ DRAM to prioritize system stability over tiering efficiency.

³Promotion from CXL back to DRAM is configured statically (§4.4.5), but the overall footprint management remains dynamically optimized.

management, avoiding the need for complex, dynamic adjustments. Keeping data in the CXL tier is strictly beneficial compared to SSD eviction as long as CXL capacity allows. It ensures higher performance and benefits SSD endurance by avoiding unnecessary writes, all without hurting the primary goal of maximizing DRAM savings. This subsystem evicts pages only when the CXL tier reaches its target utilization. The separation transforms a complex, global optimization problem across three tiers into two distinct tasks with simplified goals and constraints, significantly reducing complexity.

Note that for both subsystems, we decouple the system’s *control path* from its *data path* by configuring “migration thresholds” for data path. These thresholds can be dynamically tuned (e.g., the page coldness threshold in the DRAM $\rightarrow$ CXL path) or statically configured (e.g., the CXL utilization threshold on the CXL $\rightarrow$ SSD path). The control path does not adjudicate individual page migrations; it focuses on the high-level logic of adjusting these thresholds based on shifting workloads or system pressure. The data path then uses these thresholds to make fast, autonomous, per-page decisions without synchronous control path involvement.

Single Control Knob

Our final major system design simplification is establishing a single dynamic control knob that governs (directly and indirectly) the entire system. Concretely, this implies two key design choices.

First, Pensieve deploys a static threshold for CXL $\rightarrow$ DRAM promotion. Instead of allowing both the demotion (DRAM $\rightarrow$ CXL) and promotion (CXL $\rightarrow$ DRAM) paths—which exert opposing forces on the system—to be dynamically adjusted, we restrict the promotion path to a static, pre-configured criterion. This prevents “knob contention,” a scenario where two dynamically adjustable, opposing paths “chase” each other, leading to performance oscillations and page “ping-ponging.” By locking one side of the migration loop, we ensure the controller remains predictable and converges quickly, even under volatile workloads.

Second, Pensieve uses *indirect* control for the CXL $\rightarrow$ SSD path. Instead of avoiding the coupling between tiers, we actively leverage it based on a key observation: although the CXL $\leftrightarrow$ SSD control subsystem has fewer explicit degrees of freedom (as demotion is capacity-driven and promotion is mandatory), we can *indirectly* tune its aggressiveness by adjusting the DRAM $\rightarrow$ CXL demotion threshold. Because both loops share the CXL tier, aggressive demotion from DRAM to CXL pushes CXL utilization over target, which in turn forces more frequent CXL $\rightarrow$ SSD evictions. Conversely, reducing the aggressiveness of DRAM $\rightarrow$ CXL demotions preserves CXL headroom, delaying or reducing the volume of data swapped to the SSD. Effectively, the CXL tier acts as a pressure valve: by controlling the “inflow” (DRAM $\rightarrow$ CXL), the system naturally controls the “outflow” (CXL $\rightarrow$ SSD), thereby regulating the performance penalties from SSD accesses and reducing SSD write traffic. We evaluate the effectiveness of this indirect control in §4.6.3.

A single control knob offers several advantages. First, it drastically reduces the decision space—the system only needs to tune one knob rather than orchestrating the adjustment of multiple coupled knobs and managing their complex interactions. But more importantly,

single-variable control systems are extensively studied in control theory, allowing us to apply well-established algorithms with proven stability and robustness guarantees (as shown in §4.6.2).

4.4.3 Indicator Metrics

Pensieve ensures system stability and prevents excessive performance degradation by continuously monitoring a set of *indicator metrics* that evaluate the performance impact of the secondary memory tiers as well as system component utilization. The controller uses these indicators to dynamically adjust the aggressiveness of tiering, ensuring the system remains within an acceptable operational range.

To be effective in a production environment serving a broad spectrum of workloads, these indicators must provide a universal and low-overhead approach for monitoring performance and memory behavior. As a result, we focus on system-level metrics that can be measured in a consistent, application-agnostic manner. They must serve as effective leading signals of impending performance degradation or system instability, allowing the controller to react promptly before actual SLO violations occur.

Next, we outline the indicator metrics for Pensieve. These metrics are designed to monitor performance degradation caused by tiering and to identify system resource contention. Pensieve is extensible, supporting the integration of additional metrics or more effective alternatives.

STAR: Measuring CXL Performance Loss

Pensieve uses secondary tier access ratio (STAR) [55] as an indicator to estimate performance loss due to accesses to the CXL tier. STAR is measured as the ratio of an application’s memory bandwidth usage that goes to the slower CXL tier versus the total memory bandwidth usage; a higher value correlates with higher memory stall times and greater performance loss. We chose STAR because it imposes negligible overhead to measure online and has been demonstrated as an effective indicator of tiering performance loss in hyperscaler environments [55]. We choose the STAR limit by profiling a set of representative workloads and empirically correlating this value with acceptable levels of performance degradation.

CXL Bandwidth: Avoiding Latency Spikes

CXL performance is highly dependent on bandwidth utilization: as CXL bandwidth nears saturation, access latency increases non-linearly, causing system instability [212]. Since CXL interconnect bandwidth is shared by all four data migration paths, in addition to the direct memory accesses to the CXL tier measured by STAR, Pensieve must monitor absolute system bandwidth and prevent the system from reaching the saturation point. To determine the appropriate threshold, we profiled the CXL bandwidth-latency curve on our testbed. We define the usage limit as the point where CXL access latency increases

by 50%. This threshold naturally aligns with the “knee” of the bandwidth-latency curve, ensuring the interconnect remains in a well-behaved, predictable state.

PSI: Avoiding Excessive Page Faults

We monitor Pressure Stall Information (PSI) [203, 216]—the time tasks spend stalled while waiting for SSD page-ins—to guard against the performance degradation due to excessive page faults. We chose PSI over other metrics (e.g., page fault rate, or normalized promotion rate [119]) because it accounts for two critical factors. First, not all page faults reside on the critical path: many do not result in meaningful stalling. Second, the cost of a page fault is non-uniform, shifting with SSD hardware capabilities and transient I/O contention. We determined the PSI limit empirically by profiling a set of representative workloads, similar to our approach for selecting the STAR threshold. Because PSI directly measures the resulting stall time, it inherently accounts for the variations in SSD performance and does not require a separate metric to account for I/O contention.

SSD Write Rate: Limiting SSD Wearout

Finally, we must monitor the *SSD write rate* to ensure it does not exceed the hardware’s write endurance budget, as discussed in §4.3.1. Unlike the previous metrics, which impose instantaneous performance constraints ($\mathcal{P} \leq T_i$) that must be satisfied at any given time, the SSD write endurance budget ($\mathcal{E}$) is a *cumulative constraint* that needs to be observed over the SSD lifetime. To bridge this gap, we deploy a leaky bucket algorithm to translate the cumulative budget into an instantaneous SSD write limit that the controller must observe. This allows the system to accommodate temporary spikes in SSD write traffic, which are common during workload phase shifts, as long as the long-term average remains within the overall endurance budget.

4.4.4 Control Algorithm

Having established the indicator metrics, we describe the control algorithm that uses these inputs to dynamically adjust tiering behavior. By design, the algorithm outputs a single value: the page coldness threshold for DRAM $\rightarrow$ CXL demotion. This approach significantly simplifies the controller by eliminating the need to orchestrate and coordinate multiple, interdependent control decisions.

While the system uses a single control knob, it must respond to multiple indicator inputs, which represents a classic **Multiple-Input Single-Output (MISO)** control problem [189]. Following best practices in control theory, we decompose this problem into multiple independent **Single-Input Single-Output (SISO)** loops [189]. Each SISO loop monitors a specific indicator and independently calculates a candidate value for the coldness threshold. To ensure that all system constraints are simultaneously satisfied, the controller actuates the knob using the *most conservative* value produced by these loops. This “min-win” approach ensures that if any single indicator signals impending instability,

the system proactively throttles demotion aggressiveness, even if all other metrics remain within safe bounds.

Individual Control Loop Design

For each individual loop, we employ a Proportional-Integral-Derivative (PID) controller [242]. The controller calculates an error term by comparing the current indicator value against its limit, and adjusts the control output accordingly. PID controllers are well suited for this task as they are simple to implement and offer strong stability guarantees [14].

However, careful consideration must be given to the choice of the PID controller’s output variable (the *actuator*). While the system ultimately relies on a page coldness threshold to execute migrations, using this threshold directly as the PID output is problematic. A PID controller is designed to control continuous systems—where the system state can be continuously and differentially changed by the output. However, some coldness metrics, such as page age [119], are coarse-grained and discrete. This, coupled with the continuous nature of the PID controller, can cause the system to oscillate and fail to converge.

To achieve the necessary precision, we instead use the **Target Offload Ratio**—the ratio of memory offloaded from DRAM relative to the total memory usage of the application—as the controller’s primary actuator. This allows the PID controllers to operate on a continuous, fine-grained variable. The system then translates this ratio into the actual number of pages to demote (ranking and demoting the coldest pages first, and progressively moving towards warmer pages) until the target ratio is met. Compared to directly actuating a coarse-grained coldness threshold, this design allows the system to halt demotions at a fractional interval within a single coldness bin once the target volume is achieved. This ensures smooth, stable adjustments even when the underlying hardware or OS coldness metrics are discrete and slow-moving.

4.4.5 Static Data Path Configuration

The other data migration paths are not dynamically adjusted like DRAM → CXL demotions, but they must still be carefully configured to ensure cohesive and robust system behavior.

Watermark-Based CXL → SSD Demotions

We govern CXL → SSD demotions using three static watermarks for free CXL capacity: *low*, *high*, and *min*. Demotions trigger at the *low* watermark and continue until free capacity recovers to the *high* watermark. This dual-watermark hysteresis prevents page thrashing between the CXL and SSD tiers during minor capacity fluctuations.

The *min* watermark serves as an emergency stop, rarely triggered under normal operation. If reached, Pensieve completely halts DRAM → CXL demotions, reserving the remaining CXL space exclusively for mandatory SSD → CXL promotions. We sized the threshold based on CorpX production telemetry to safely cover the 99th percentile of sudden, per-minute page fault bursts.

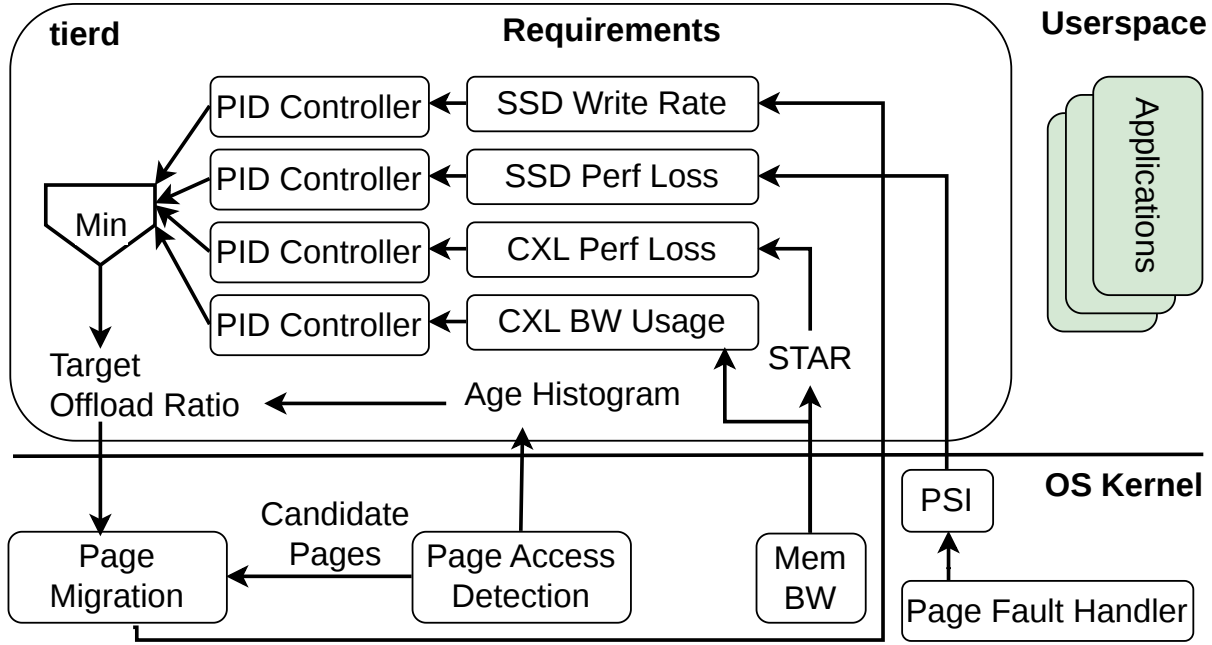


Figure 4.2: Pensieve system overview.

Access-based CXL → DRAM Promotions

We statically configure the CXL → DRAM promotion criterion to select pages that are accessed in two consecutive observation windows. Requiring accesses across two consecutive windows effectively filters out the “one-hit wonder” pages that are only fleetingly accessed [145]. Choosing a short window size allows the system to react promptly, minimizing the performance penalty of accessing newly hot pages from the slower CXL tier. To strike the right balance, Pensieve uses a 5-second observation window.

Mandatory SSD → CXL Promotion

Accessing a swapped out page on SSD results in a mandatory promotion via a page fault. By default these promotions strictly target the CXL tier (§4.4.2). We do, however, allow an emergency fallback path: if the CXL tier has no free space and cannot accommodate the incoming page, the page is promoted directly from the SSD to DRAM. This emergency bypass avoids fatal out-of-memory errors and crashes, improving system reliability under extreme memory pressure.

4.5 Implementation

Figure 4.2 shows the system architecture of Pensieve. It consists of two components: a userspace daemon (**tierd**) and a set of kernel mechanisms. **tierd** handles the policy and control aspects of the system (≈ 2100 lines of C++ code). It takes the indicators (measured

by the kernel) as inputs, runs the control algorithm every 5 seconds, and outputs the target ratio of pages to be offloaded from DRAM, which is then executed by the kernel. `tierd` incurs minimal runtime overhead (2% of one CPU core and 60MB of memory).

The kernel mechanisms are implemented on top of a fork of upstream Linux kernel 6.6. Importantly, this kernel runs a global page access detection thread, which periodically scans the PTEs and increases the age of any un-accessed page. We modify the kernel to implement the data migration paths based on the migration thresholds. The kernel also collects the indicators metrics (PSI, STAR, SSD write rate, and memory bandwidth usage) and passes them to `tierd`.

4.5.1 Missing Linux Support for Three-Tier Memory

While Linux can support both CXL (as a CPU-less NUMA node) and SSD tiers, it lacks mechanisms to explicitly control the data flow between these tiers at a fine granularity. First, Linux does not support controlling demotion from the CXL and DRAM tier separately. To address this, Pensieve introduces fine-grained interfaces to independently control demotion from the DRAM and CXL tiers, allowing the system to precisely specify the number of pages to demote from each. Second, when a page is swapped in, Linux does not provide control over its placement. Instead, it relies on a hard-coded policy of placing the page in the local NUMA node, falling back to remote NUMA nodes if full. Pensieve allows the system to explicitly specify the primary and fallback destination tiers, offering precise control over the promotion path. Finally, Pensieve implements a customized *PSI* metric, which more accurately accounts for page-fault-induced performance loss in the highly parallel applications we support than the existing *PSI* metric Linux offers [203].

4.5.2 Flexible Coldness Tracking

Pensieve is designed to support different page coldness tracking mechanisms (e.g., page aging [55, 119], LRU lists, or custom hardware mechanisms [39]) as long as they can be translated into a per-page coldness/hotness indicator, which we use for migration threshold comparisons. Our current implementation uses access bit-scan based page aging, tracking the number of scans for which a page is inactive as the coldness indicator. However, Pensieve can accommodate emerging hardware-assisted telemetry to lower CPU overhead and increase the resolution of hot/cold information [39].

4.6 Evaluation

We evaluate Pensieve to address three key questions: (1) Can Pensieve achieve greater DRAM savings than two-tier DRAM+SSD or DRAM+CXL systems while maintaining target performance? (2) Can Pensieve remain stable under dynamic workloads and varying operating conditions? (3) Are our key design choices well-founded?

Testbed We evaluate Pensieve on a dual-socket Intel Granite Rapids CPU server equipped with 768 GB DDR5 DRAM, 384 GB DDR4-backed CXL memory [39], and a 150 GB SSD swap partition [192]. The CXL memory subsystem delivers a peak bandwidth of 35 GB/s with an unloaded latency of 360 ns.

By default, we set the following CXL utilization watermarks: min (2%), low (10%), and high (15%). We set performance loss thresholds to 10% STAR and 0.5% PSI, which empirically maintain acceptable application performance. To ensure longevity and stability, we limit the SSD endurance write budget to 8 MB/s and cap CXL bandwidth at 31 GB/s. At this bandwidth, CXL latency increases by 50% over the unloaded baseline.

Baselines We compare Pensieve against DRAM-only, DRAM+CXL, and DRAM+SSD configurations. The DRAM+CXL baseline employs a feedback-driven control loop to regulate *proactive demotion* from DRAM to CXL, using STAR as a performance indicator—consistent with the design in [55]. The DRAM+SSD baseline uses a similar control loop for *proactive reclamation* driven by PSI [216]. We exclude default Linux two-tier configurations as they lack proactive mechanisms to free up DRAM, a core requirement of our system.

Table 4.1: Summary of evaluated applications in Pensieve.

App	Description	Metric
Memcached [60]	In-memory object cache; the mean of request keys shifts over time.	Req / sec
GAPBS [27]	Betweenness Centrality.	Time
Mediawiki [174]	Web serving (language runtime, reverse proxy, database, cache).	Req / sec
DB	Distributed relational DB at Meta.	Req / sec
RB [185]	Emulates hotness distribution.	None

Workloads Table 4.1 summarizes our evaluation workloads, which are memory-intensive and representative of real-world data center applications. Open-source workloads cover caching, graph processing, and web serving. DB is a distributed relational database in production at Meta. RB [185] emulates the page hotness distribution from hyperscalers’ production traces; it does not report a performance metric. Because total memory capacity in our testbed exceeds the memory footprint of our evaluated workloads, we emulate a 3:1 DRAM:CXL ratio (consistent with hyperscaler deployment patterns [145]) by scaling the available CXL tier to 25% of the footprint and disabling remaining physical memory. For baselines without CXL, we ensure DRAM capacity fits 100% of the workload footprint. Since DRAM savings fluctuate during execution, we report the average savings achieved over three independent trials.

4.6.1 DRAM Savings and Performance

Pensieve consistently achieves greater DRAM savings than two-tier baselines while remaining within performance loss limits across applications with diverse memory access

patterns. Figure 4.3 shows the DRAM savings and performance degradation for Pensieve for all systems relative to DRAM-only. Pensieve delivers on average 33% DRAM savings across workloads, substantially outperforming DRAM+CXL (21%) and DRAM+SSD (18%) two-tier systems.

DRAM+CXL savings are strictly capped by the provisioned CXL capacity (25%). Pensieve bypasses this constraint, achieving 33–44% savings (excluding GAPBS) while adhering to our 5% performance loss target. Although DRAM+CXL maintains lower performance overhead—since all migrated pages reside in the more performant CXL tier—it fails to meet our primary design goal of maximizing DRAM savings.

Compared to DRAM+SSD, where DRAM savings range from 15%–38% (excluding GAPBS), Pensieve consistently delivers superior savings with lower performance loss. While DRAM+CXL is capacity-constrained, DRAM+SSD is primarily limited by the performance degradation of the SSD tier. Pensieve alleviates this bottleneck by effectively utilizing the intermediate CXL tier, enabling higher DRAM savings without the steep performance penalties of SSD-only tiering.

Among the evaluated workloads, GAPBS is a unique edge case: its limited cold memory fits within the CXL tier and incurs negligible performance impact when offloaded. For such workloads, a three-tier system offers no marginal benefit over two tiers. In this case, Pensieve correctly defaults to two-tier behavior, offloading all cold memory without additional overhead. Our analysis of Meta’s workloads suggests GAPBS is an outlier though. Most data center workloads contain significant cold data—prior studies identify an average of 35% (and up to 62%) of memory as idle [145, 216]. This presents a substantial opportunity for Pensieve to improve DRAM efficiency, breaking the limits of a single secondary tier.

Performance vs. Savings Trade-offs To investigate the relationship between DRAM savings and performance loss, we analyze *memcached* and *mediawiki* by varying their STAR and PSI limits. Figure 4.4 presents the resulting Memory-Saving Performance Curve (MPC), which maps achieved DRAM savings against application performance loss. Curves shifting toward the top left represent superior efficiency, achieving higher DRAM savings with minimal performance impact.

Figure 4.4 illustrates the MPCs for *memcached* (top) and *mediawiki* (bottom), showing that Pensieve consistently outperforms two-tier baselines. While DRAM+CXL (blue) offloads memory with minimal performance degradation (under 2%), it is strictly capped by the CXL physical capacity, peaking below 25% due to CXL tier watermarks (§4.4.5). Pensieve bypasses this limit by swapping the coldest pages to SSD to demote warmer pages to CXL, reaching 49% DRAM savings at an 8.5% performance loss for *memcached*. Although the DRAM+SSD baseline can increase savings by accepting higher performance degradation, it is constrained by the SSD write endurance limit at 24% savings. *Mediawiki* confirms this trend: while DRAM+SSD achieves comparable savings to Pensieve (38% vs. 41% for Pensieve), it incurs a significantly higher performance cost (7.2% vs. 3.5%). Across all thresholds and workloads, Pensieve delivers superior DRAM savings by effectively leveraging the three-tier hierarchy.

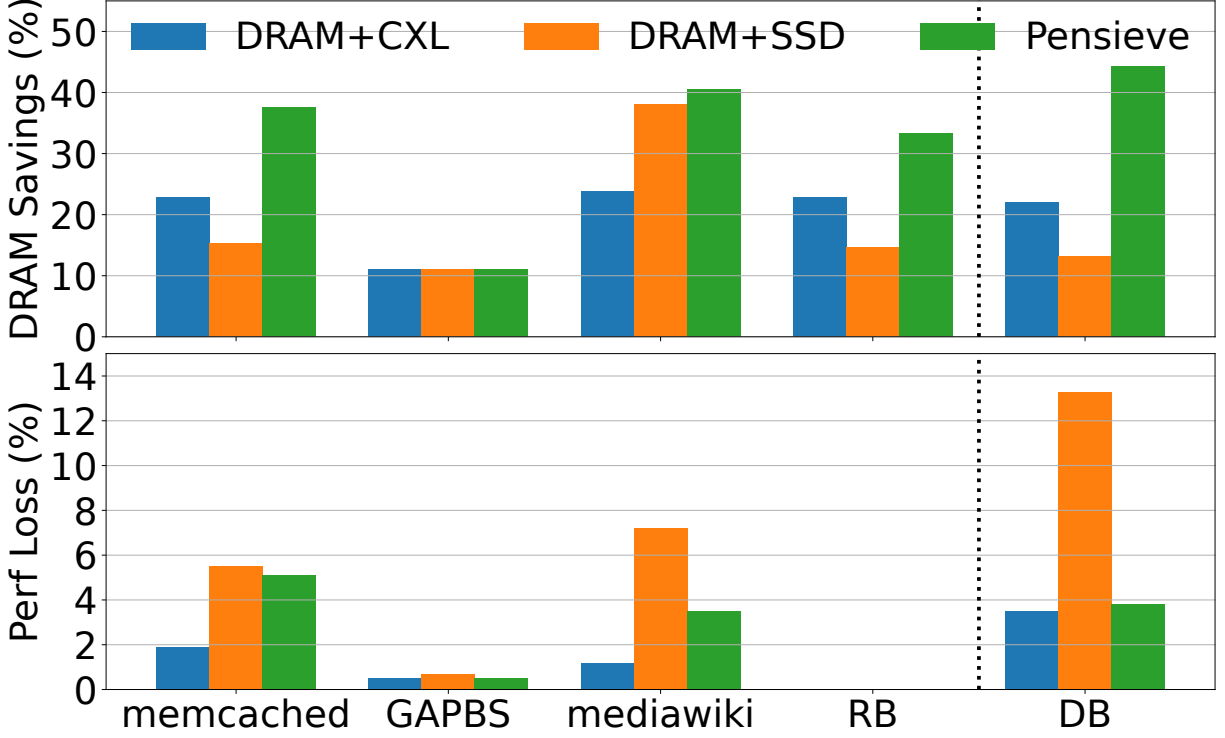


Figure 4.3: DRAM tier savings and performance losses measured by application-specific metrics of two-tier systems and Pensieve. To the right of the dotted line is a Meta production workload. RB does not report performance.

4.6.2 System Stability

Dynamic workloads Pensieve maintains stability within target operational ranges even under high workload churn. We evaluate this using *memcached* with a shifting “hot set”—a normal key distribution whose mean moves every 10 minutes, forcing the system to continuously adapt. The workload comprises 3.5 M keys (2–8 KB each). Figures 4.5 and 4.6 depict the resulting memory usage timeseries and the corresponding STAR and PSI metrics.

Following a five-minute initialization, Pensieve begins offloading DRAM. By minute 20, memory utilization across all tiers stabilizes, offloading 6 GB of DRAM while fully utilizing the high-performance CXL tier. While the system initially overshoots STAR and PSI thresholds, it recovers by promoting pages from the SSD and CXL tiers back to DRAM. By minute 50, PSI and STAR converge at their predefined limits, maximizing DRAM savings without compromising performance. As the *memcached* access pattern shifts every 10 minutes, both metrics momentarily spike. Pensieve responds by identifying the active constraint (one of STAR, PSI, and swapout rate), using the deviation to throttle offloading and redistributing pages across the tiers to accommodate the new working set. Figure 4.7 illustrates these active constraints or the *normal* state when all are satisfied. This experiment demonstrates the resilience of Pensieve’s control logic during dynamic

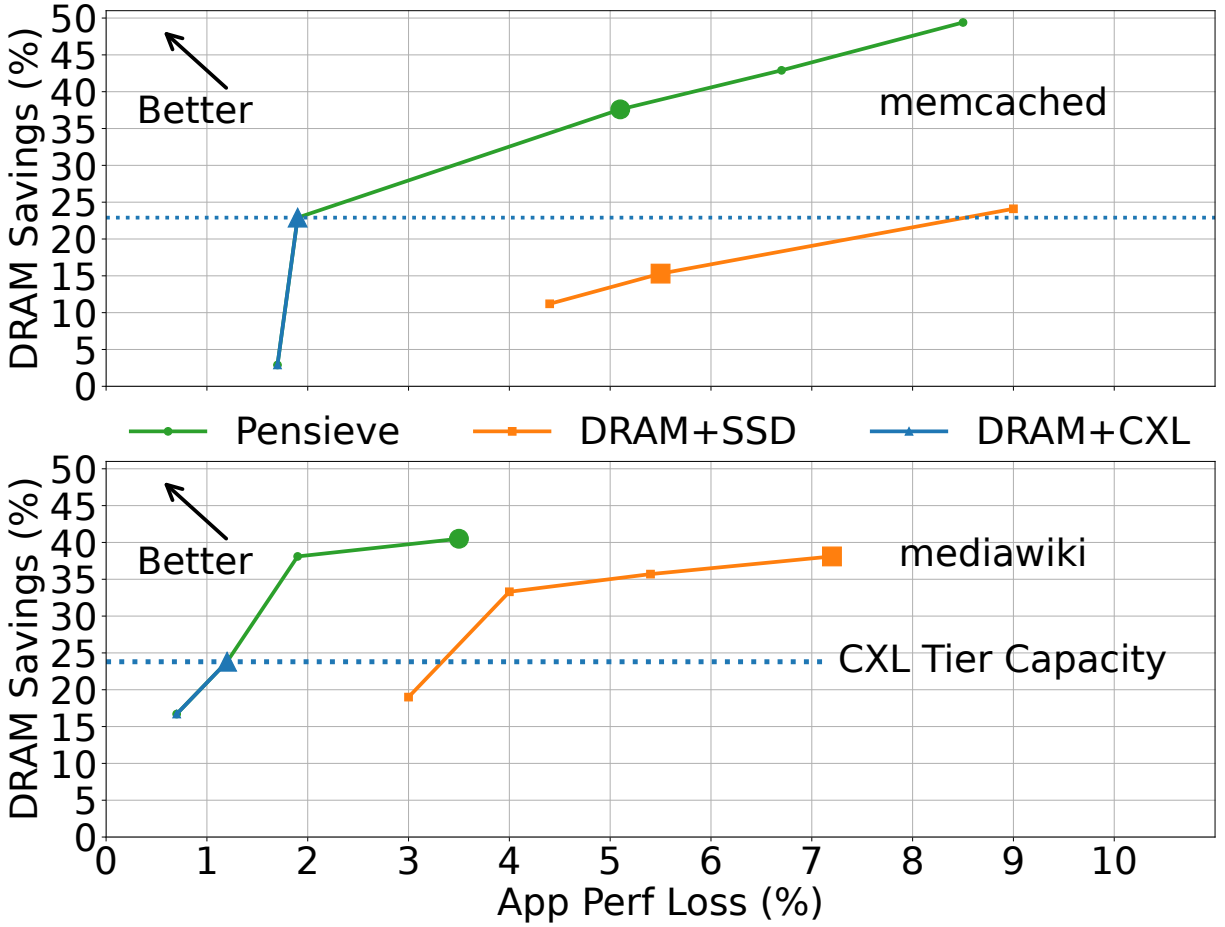


Figure 4.4: DRAM savings at various application-reported performance losses, comparing Pensieve and two-tier systems using memcached (top) and mediawiki (bottom). The larger dots show the operating points of Figure 4.3.

workload transitions.

CXL bandwidth contention Because memcached alone cannot saturate CXL bandwidth, we use Intel MLC [94] to consume 31 GB/s of CXL bandwidth in the background. We then compare Pensieve against a variant that lacks an explicit CXL bandwidth constraint. Without this constraint, Pensieve still respects STAR and PSI targets and achieves the same DRAM savings reported in § 4.6.1, yet memcached throughput drops by 5.1%. This occurs because CXL latency spikes near saturation; at these levels, identical STAR values translate to significantly higher performance degradation. This justifies the inclusion of a CXL bandwidth constraint to ensure predictable application performance.

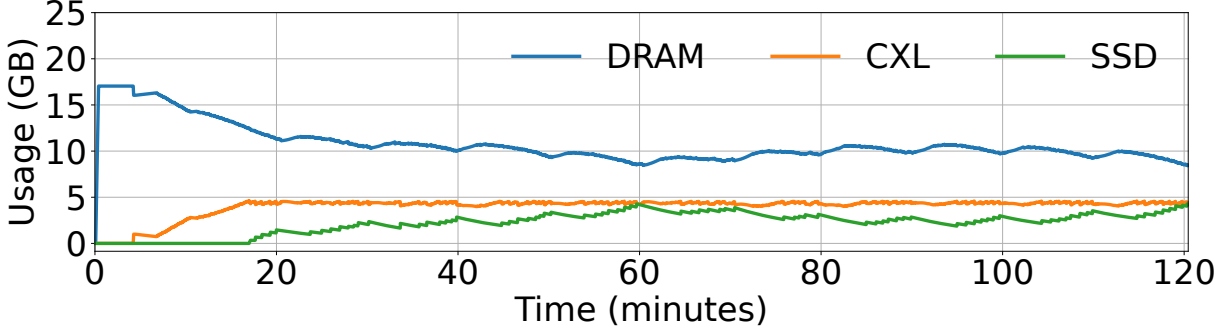


Figure 4.5: Timeseries of memory usage of memcached.

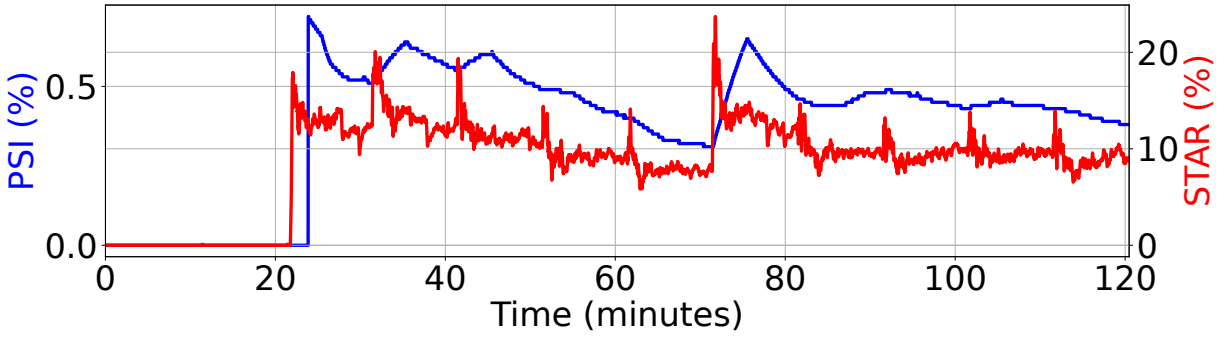


Figure 4.6: Timeseries of STAR and PSI of memcached.

4.6.3 Validating Design Decisions

Hierarchical Data Path

Pensieve organizes the three tiers as a hierarchy and allows only page migrations between adjacent tiers during normal operations. To understand the effects of this decision, we compare to a modified system where promotions from SSD go to DRAM, as in many existing memory tiering solutions [145]. We see that if swapped in pages go directly to DRAM, the achieved DRAM savings drop by 4% compared to our design (34% vs. 38%), while their performance differences are within the margin of error. This is because of a constant stream of non-hot pages landing in DRAM, which take time to age and be demoted to lower tiers, inflating the DRAM usage. To confirm this theory, we further analyze the subsequent accesses to the swapped-in pages during the experiment. We find that 51% of swapped in pages become cold again after 30 seconds, being poor candidates to be promoted directly to the DRAM tier. In conclusion, Pensieve’s hierarchical data paths are beneficial in filtering promotions through the CXL tier to avoid inappropriate page placement.

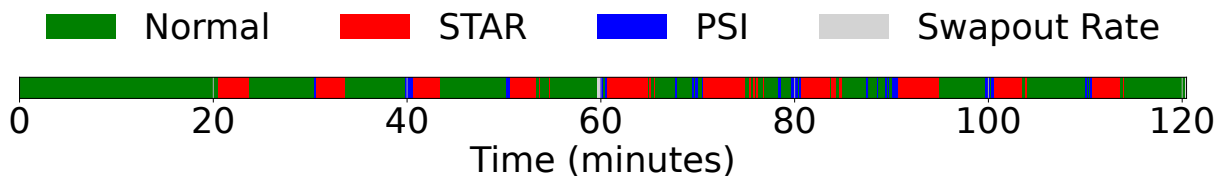


Figure 4.7: Timeseries of active constraints of memcached.

Unified Control

Pensieve manages the DRAM $\leftrightarrow$ CXL and CXL $\leftrightarrow$ SSD data paths in one unified control loop. In §4.3.2, we attempted stacking two independent control loops on top of each other, and observed low CXL tier utilization. We extend the configuration by adding a CXL utilization constraint: CXL $\rightarrow$ SSD demotions happen when both PSI is under limit and CXL utilization is high. We still observe unsatisfactory results, because when PSI is high, CXL $\rightarrow$ SSD demotion will stop, blocking further DRAM $\rightarrow$ CXL demotions and reducing the DRAM savings to 32% compared to 38% under Pensieve. Furthermore, a high PSI may leave no space in CXL for SSD $\rightarrow$ CXL promotions: in fact, we observe numerous instances of uncommon case SSD $\rightarrow$ DRAM promotions (§4.4.2). Although further optimizations to resolve more couplings are possible, our attempts highlight the fragility of using two separate control loops with independent control knobs and the additional tuning required to make them work well.

Indirect Control

We evaluate whether Pensieve’s single control knob of DRAM $\rightarrow$ CXL demotion threshold is effective for controlling **all** data paths. The experiment uses memcached plus an I/O test application [21] that generates antagonistic I/O requests to the SSD device, starting the I/O test once a significant number of memcached pages are offloaded to CXL and SSD. With the SSD device under heavy use, page faults incur significantly higher latencies, driving memcached’s PSI value to 28% and illustrating that PSI is responsive to dynamic SSD performance. Pensieve senses the PSI violation and increases the threshold for DRAM $\rightarrow$ CXL promotions. In response, DRAM $\rightarrow$ CXL demotions slow immediately, followed by CXL $\rightarrow$ SSD demotions after about one minute, reducing DRAM savings. Once a sufficient number of pages are promoted from the SSD tier to bring PSI below its limit, the demotion pipeline resumes. This experiment shows that Pensieve can effectively manage the entire tiering hierarchy using a single control knob.

4.7 Related Work

Memory Tiering: Extensive prior work addresses memory tiering, including page migration strategies [121, 212], performance modeling [55, 77, 128, 129, 216], hardware support [238, 240], and page sizing [2, 121]. Most relevant to Pensieve are systems designed for data center memory overcommit that *proactively* reclaim top-tier DRAM by offloading

memory to a secondary tier, such as compressed memory [126, 216], SSDs [216], NVM [55], or CXL [145]. However, these remain strictly two-tier architectures. TMO [216], for instance, statically assigns either compressed memory or SSD as the sole secondary tier per application. In contrast, Pensieve operates as a true three-tier system.

Performance Indicator Metrics: To quantify application degradation caused by memory tiering, Pensieve relies on online performance indicators. While recent work has proposed various system-level metrics [77, 128, 129], those requiring offline profiling [128, 129] are incompatible with Pensieve’s dynamic, online control loop. Pensieve is modular and can integrate new online indicators as they emerge.

Multi-tier Storage Systems: Pensieve conceptually resembles multi-tier storage systems [45, 72, 103, 114, 118, 147], which proactively migrate inactive data to slower, cheaper tiers (e.g., HDD or tape) while reserving faster tiers for active data. Pensieve diverges from traditional storage tiering in two critical ways. First, while lower storage tiers tend to have higher capacity, Pensieve is explicitly capacity constrained by the CXL tier, necessitating careful management of tier interdependencies. Second, storage systems can directly observe the access frequency and overhead of data under their management since they operate on the path of access. In contrast, Pensieve must use performance indicators to measure the overhead of accessing data on the slower CXL tier, requiring careful selection of such indicators.

Control Theory in Systems Management: PID controllers have been widely deployed in systems design to ensure operational stability under highly dynamic conditions [81, 153, 165, 214] across diverse environments [14, 242]. Pensieve draws from foundational principles in control theory to simplify a complex, highly coupled Multi-Input Multi-Output (MIMO) environment into a manageable MISO model [81, 189]. This deliberate simplification of the three-tier control path avoids the instability of interfering control loops, achieving predictable and low-overhead management.

4.8 Conclusion

Pensieve is the first three-tier memory system that combines swap-based and byte-based memory tiers in a single system to solve the data center problem of reducing DRAM usage while meeting performance and system-level constraints. It tames the complexity and coupling challenges of three-tier systems using a hierarchical data migration path, decoupled DRAM $\leftrightarrow$ CXL and CXL $\leftrightarrow$ SSD management, and a unified dynamic control knob for the entire system. Pensieve achieves significantly greater DRAM savings than two-tier systems alone, while maintaining acceptable application performance degradation, respecting SSD endurance constraints, and maintaining stable operations even under dynamic workload shifts and environmental changes.

Chapter 5

Equilibria: Fair and Multi-Tenant CXL Memory Tiering At Scale

5.1 Introduction

The rise in memory demands of data center applications has led to increased spending on memory in data centers, including power consumption and capital expenditures [120, 152]. Memory can account for nearly a quarter of the rack power consumption and up to 44% of the Total Cost of Ownership (TCO) of a typical compute server in Meta’s data centers. Memory expansion based on CXL [46] has emerged as an effective way to mitigate the cost and power challenges. CXL is a cache-coherent interconnect that can attach memory backed by a multitude of technologies to the host [102, 145, 222, 227]. At Meta, recycled older-generation DRAM is connected to the host via CXL, providing 20–30% of a server’s total memory capacity, yielding significant cost savings. We refer to CPU-attached DRAM as *local memory* and CXL-attached DRAM as *CXL memory*, and they are in two separate NUMA nodes.

Despite its potential benefits, CXL memory introduces higher access latency, making software support essential to prevent performance degradation. While applications may explicitly manage allocations across NUMA nodes, our interactions with multiple product teams within and outside Meta reveal a strong preference for transparent memory-tiering mechanisms that abstract away this complexity. At the same time, data centers are increasingly adopting multi-tenancy, where large servers are shared by multiple colocated workloads. Multi-tenancy is a highly effective strategy for maximizing the utilization of expensive hardware and improving performance per TCO. By colocating workloads with complementary resource demands, multi-tenancy enables more balanced and efficient resource utilization. This approach not only boosts the performance of individual compute nodes and reduces the number of servers required, but also enhances overall fleet efficiency, ultimately improving cost-effectiveness and scalability across data centers.

Prior work [2, 16, 41, 42, 55, 110, 121, 123, 127, 145, 173, 184, 196, 212, 223, 226, 238] has simplified the use of CXL memory through memory tiering, where hotter pages are transparently migrated to local memory and colder pages to CXL memory. However,

multi-tenancy introduces new fairness challenges for deploying CXL memory at scale that have not been identified or examined by prior work. In this work, we for the first time systematically uncover and characterize these challenges using insights from real production deployments at Meta.

First, hyperscale services have diverse requirements for performance guarantees and service level objectives (SLOs), which complicates the job scheduling due to performance uncertainties. Existing memory tiering solutions lack multi-tenancy support and cannot guarantee fair memory sharing across tiers among colocated tenants. They primarily optimize for system-level metrics such as putting the system-wide hottest or most performance-critical pages into local memory, ignoring fairness between colocated workloads. Even when they do consider multi-tenancy, they rely on coarse categorizations [55, 123] of workloads based on their importance such as *critical/best-effort* for scheduling jobs on tiered memory systems. This ignores the problem that colocated "critical" workloads need to meet their own performance SLOs but still suffer from noisy neighbor effects. In fact, at Meta, the vast majority of multi-tenancy setups are multiple instances of the same workload with SLO requirements stacked together. This results in performance variability across containers and leads to severe SLO violations. For example, when two workloads with different memory access patterns are colocated, one may receive the majority of the local memory while the other receives very little under a conventional tiering scheme. For another example, in a multi-tenant environment where tenants dynamically arrive and depart on hosts, workloads that launch later are disadvantaged, as they are forced to rely more heavily on CXL memory after local memory has been exhausted. As we will demonstrate in Section 5.3 and 5.5, the lack of a fairness mechanism in colocating workloads leads to performance losses of up to 65% for Meta’s production workload *Cache*, making CXL memory difficult to deploy for production workloads.

Second, operations in memory tiering such as demotion, promotion, setting up NUMA hint faults [2, 231], hint faults triggering, and others, are not virtualized across tenants (through containers or VMs). This makes it impossible to monitor the performance and tune the behavior of memory-tiering for colocated workloads, which is essential in today’s data centers that run first-party workloads.

Finally, most existing solutions are not thrashing-aware. We find that if a workload is thrashing between local and CXL memory, it can cause serious interference to colocated workloads and slow down the entire system. Therefore, detecting and mitigating thrashing is a prerequisite for a production-grade multi-tenant memory tiering solution. All of these issues make CXL memory adoption in multi-tenant environments challenging, and new solutions are needed to pave the way for CXL memory in data centers.

We build upon our learnings from deploying CXL memory in production and propose Equilibria as the first holistic solution for fair multi-tenant CXL memory tiering in data centers. Equilibria is a solution in the OS and userspace that has four main components: (a) multi-tenancy-aware controls and observability for tiered memory – interfaces that enable users to configure memory placement policies and monitor tiered memory utilization and operations per tenant; (b) automatic memory fair share determination for a set of jobs scheduled on a server by online profiling and solving an optimization problem of CXL memory capacity, access, and bandwidth; and (c) demotion and promotion regulations

that enforce the fair memory placement and meet the SLOs of colocated application; and (d) thrashing mitigation that reduces the adverse effects of a thrashing workload to the whole system.

We make the following contributions in this work:

1. We share our learnings from deploying CXL memory in production in Meta’s data centers. They include the latest trends of memory hardware, bandwidth, application adaptation, as well as the acute fairness issues that are exposed when multi-tenancy is added to the equation. To our best knowledge, we are the first to share observations from a real-world CXL memory deployment in data centers with multi-tenancy enabled.
2. We design and implement the first solution for fairness in multi-tenant CXL memory deployments, Equilibria, which provides per-container tiering observability, memory fair share determination and controls, and thrashing mitigation that help colocated workloads meet their SLOs.
3. We evaluate Equilibria with benchmarks and Meta’s production workloads. Equilibria meets the SLOs of colocated applications and avoids performance anomalies, providing up to 52% and $1.7\times$ performance improvements across production and benchmarks respectively over baseline Linux/TPP [145].
4. Equilibria is being deployed to the entire fleet of Meta. We have shared Equilibria patches with the Linux community, a subset of which has been accepted upstream.

5.2 Background

5.2.1 Memory Costs in Data Centers and CXL Memory

Memory costs and power overheads have become a significant challenge in designing efficient systems for data centers. Figure 5.1 shows the contribution of memory towards the overall rack-level cost and power for one compute server types at Meta. Over the last decade, memory has grown to be one of the biggest contributors towards rack-level capital expenditures (CapEx) and operating expenses (OpEx) [123, 145]. The reasons for this trend are the following. First, workloads are dealing with more data, hence memory bandwidth and capacity requirements per unit of work have grown over time. Furthermore, memory technologies have not been scaling at the same rate as the other hardware components.

Compute Express Link (CXL) presents a promising approach to addressing this challenge, offering greater flexibility in terms of capacity, cost, and performance characteristics. CXL is a cache-coherent interconnect that enables various memory technologies, including DDR, LPDDR, NAND, and persistent memory. CXL memory is presented as memory-only NUMA domains and has increased access latency compared to CPU-attached DRAM, which can negatively impact application performance when significant portions of memory accesses are to CXL memory. This means that software support is required to effectively use CXL memory without compromising application performance.

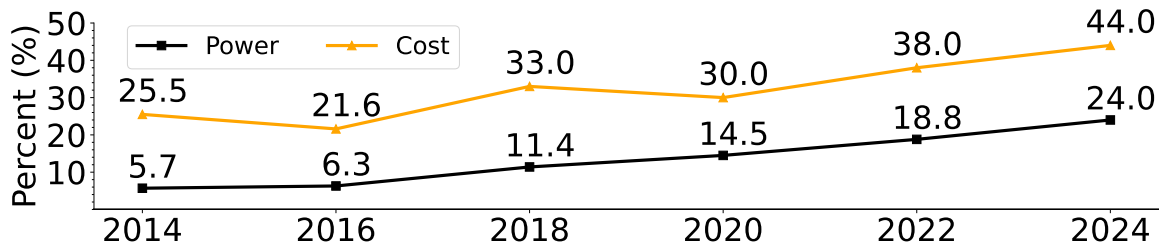


Figure 5.1: Memory as a % of rack TCO and power at Meta.

5.2.2 Containers for Multi-Tenancy

Data Centers have been moving towards larger servers in terms of cores and memory, which are multiplexed to support multiple tenants using containers or virtual machines (VMs). In fact, containers have become the de facto standard for continuously developing, testing, distributing, and deploying applications at hyperscale. Some operators choose to have an additional virtualization layer where a physical server is divided into VMs; nonetheless, containers still run inside those VMs. Such containerization of application deployment has been shown to reduce the cost of running a modern data center for the following reasons. First, larger servers have lower TCO per unit of resources due to recent hardware trends [71]. Second, having one application use up the resources of an entire server is difficult, whereas containers allow multiple workloads to better fit the shape of the server, reducing stranded resources. At Meta, multiple containers typically run side by side on the same physical server, and host agents collect per-container metrics to monitor workloads’ performance and set the resource limits of each container. Therefore, any CXL memory deployment must have full support for multi-tenant deployment with containers and provide container-level observability and configurability of memory tiering.

5.3 Learnings from Production

While CXL memory offers significant potential for reducing memory costs in data centers, the initial deployment of CXL memory in Meta’s multi-tenant data centers reveals several key challenges that must be addressed.

5.3.1 Real vs. Emulated CXL Memory

Prior work in CXL memory has primarily relied on emulating CXL memory by disabling CPU cores on multi-NUMA node machines. However, we noticed that emulated CXL memory differs from real CXL memory in the following performance characteristics. Figure 5.2 depicts the latency versus bandwidth for local, remote (across NUMA-node), and CXL-attached memory. Real CXL memory has higher idle latency (by 27 ns) and higher loaded latency (by 37 ns at 30% load) compared to emulated CXL memory, while the achievable peak bandwidth efficiency is lower. This implies that workloads running on a real CXL system will generally be more sensitive to the efficacy of memory tiering than em-

ulated systems might suggest, and highlights the importance of evaluating memory tiering solutions on real hardware.

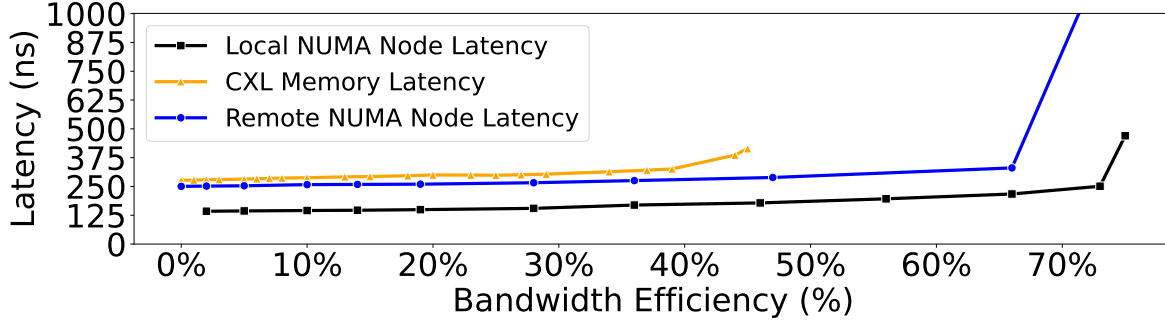


Figure 5.2: Latency versus bandwidth for different memory tiers.

5.3.2 Capacity-Bound Apps Use Little CXL Bandwidth

When running memory capacity-bound workloads, we observe that most have a significant amount of cold memory that can be placed in CXL memory without compromising application performance. Table 5.1 shows data from three large production applications at Meta running on a system with 1TB of total memory (768GB local and 256GB over CXL). Each application uses a significant percentage of CXL memory capacity, but drives a very small amount of bandwidth on the CXL bus. This shows that we do not need to provision a large number of PCIe lanes to attach CXL memory devices to get the bulk of the benefits from CXL memory expansion. So, we continue to use a single x8 or x16 PCIe controller to connect CXL memory across several generations of servers.

Application	Local Mem Used	CXL Mem Used	Local BW	CXL BW
App A	718 GB	101 GB	157 GBps	3 GBps
App B	740 GB	104 GB	40 GBps	1 GBps
App C	508 GB	151 GB	41 GBps	3 GBps

Table 5.1: Bandwidth usage of memory capacity-bound apps.

5.3.3 CXL Does Not Remove Bandwidth Bottlenecks

While CXL memory potentially offers both memory capacity and bandwidth benefits, an analysis of Meta’s production workloads and hardware indicates that additional memory bandwidth from CXL memory has limited practical use in current deployments. As shown in Table 5.2, CXL memory provides only a small fraction of total system bandwidth (10%), and the ratio of CXL to local memory bandwidth has not scaled over hardware generations. Furthermore, relying on DDR main memory bandwidth scaling is more power efficient than increasing the number of PCIe lanes to get additional memory bandwidth from CXL.

To illustrate this point further, Table 5.3 shows data from a memory bandwidth-bound production workload while using different interleaving ratios to place a portion of application memory on CXL and utilize the additional bandwidth for performance improvements. Overall, we only see up to a 2.5% performance upside on this application on a system configured to provide 10% of its total memory bandwidth on CXL (the Gen1 system from Table 5.2). Therefore, we focus on capacity expansion rather than bandwidth improvements, providing significant savings by extending memory capacity at low costs through the reuse of older DRAM.

Hardware Gen	CXL-Mem Bw.	Local-Mem Bw.	CXL-Mem: DRAM Bw.	CXL-Mem: DRAM Cap.
Gen1, Year 1	30GB/s	300 GB/s	0.1	0.25
Gen2, Year 3	60GB/s	620 GB/s	0.1	0.33
Gen3, Year 3	60GB/s	620 GB/s	0.1	0.375

Table 5.2: Bandwidth and capacity scaling of memory.

Interleaving	Local Bw. %	CXL Bw. %	Relative Perf
1:0	59%	0%	1.00
2:1	31%	48%	0.97
3:1	46%	25%	1.025
4:1	48%	21%	1.021

Table 5.3: Application performance gains from CXL bandwidth.

5.3.4 Most Applications Require Transparent Tiering

Achieving broad adoption of CXL memory for capacity expansion hinges on minimizing the burden on application developers. In working with owners of major workloads at Meta, we observed a strong reluctance to implement custom memory management logic within their applications. Determining which data is hot or cold and managing its placement across memory tiers adds significant complexity, especially given the diverse server hardware generations and types on which these applications must run. Consequently, we focus on transparent memory tiering mechanisms instead of pursuing per-application support.

5.3.5 Hardware Support for Tiering is Insufficient

Ideally, hardware might assist with transparent tiering [11, 40, 93, 122, 183, 220, 232], but current processor architectures and memory hardware have not evolved to adequately support memory tiering. For instance, Intel’s Two-Level Memory (2LM) [164, 238] provides hardware-managed tiering, but has crucial limitations. First, it requires a direct-mapped DRAM region as a cache for the CXL memory, necessitating a 1:1 hardware-managed local memory to CXL memory ratio. Secondly, it is not application-aware and cannot

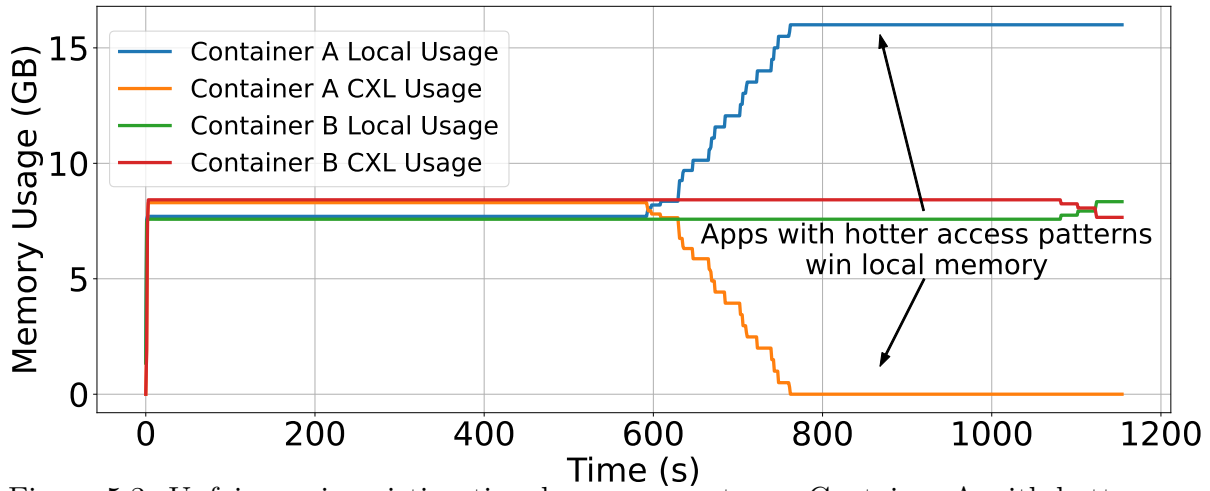


Figure 5.3: Unfairness in existing tiered memory systems. Container A with hotter access patterns gets 100% local memory, whereas Container B gets only about half.

apply per-app policies. Similarly, while CXL memory or DRAM may incorporate hotness monitoring units and autonomously perform memory tiering, such features are not widely implemented.

Given these hardware limitations, we build upon the software support in the OS to manage CXL tiered memory and ensure fairness under multi-tenancy. In the future, we envision targeted hardware support for page migration and hotness tracking that works together with the OS to manage CXL memory with less overhead and more precision.

5.3.6 Multi-Tenancy and Fairness

Existing tiered memory systems primarily make system-level decisions such as putting the system-wide pages that are hottest or most critical to performance into local memory, ignoring fairness between colocated workloads. This can cause applications with colder access patterns to suffer. To demonstrate this point, we used two instances (Container A, B) of a streaming microbenchmark that accesses memory at configurable hotness levels, where hotness is defined as the frequency of repeated memory access on a block of memory. As shown in Figure 5.3, Container A, which has twice the hotness of Container B, is able to eventually have all its memory footprint in local memory when local memory is contended. On the other hand, Container B only gets about half of its memory footprint in local memory, leading to a lower access throughput compared to Container A.

Furthermore, there are performance variations between workloads that exhibit the same access hotness solely due to their launch order. We launch two instances of the microbenchmark with the same hotness, and Container B is launched 30 seconds later than Container A. Container A allocates memory first and is able to get all local memory. When Container B starts, local memory is exhausted, so it spills to CXL memory. Then, the memory distribution remains mostly unchanged for both Container A and B, but the performance of Container B is permanently impaired with 28% lower access throughput than Container A because of its high CXL memory usage. Even though Linux attempts to keep a small amount (controlled by the *low/min watermarks*) of local memory free so that new alloca-

tions (assumed to be hot) have a higher chance of landing on local memory [145] before falling back to CXL memory, kernel background/kswapd reclaim can only free up local memory when there are inactive pages, and often cannot keep up with the speed of new allocations so they fall back to CXL memory.

Finally, if a workload is thrashing between local memory and CXL memory, existing memory tiering solutions do not limit its negative performance impacts on other colocated workloads. We launch the microbenchmark with a working set significantly larger than the capacity of local memory, causing heavy thrashing. We also run two other instances of the benchmark with non-thrashing access patterns in CXL memory. We observe that under baseline Linux, there are constant and heavy promotion and demotion activities attributed to the thrashing workload at over 100K pages per second, while the other two workloads are deprived of memory tiering management attention with less than 1/10th of the page migration activities than when they run alone, so they are stuck using mostly CXL memory and have 7% lower throughput than when they run alone. This demonstrates that without proper mitigation, a thrashing workload has global negative effects on tiered memory systems.

In summary, the lack of fairness when using CXL memory leads to performance variability, and any tiering solution must resolve the fairness issue in multi-tenant environments to be deployed in production data centers.

5.4 Equilibria Design

We propose Equilibria, a holistic solution in the OS and container runtime to ensure fairness and meet workload SLOs in multi-tenant data centers with CXL-based tiered memory.

5.4.1 Fairness: Definition and Challenges

We define fairness as each colocated workload being able to run at a performance level that meets its SLO, regardless of when it is launched and with whom it is colocated. The goal of Equilibria is to help workloads always achieve their performance targets even with diverse colocated workloads and varying job schedules. Equilibria focuses on memory placement of colocated workloads and guarantees that a certain ratio of an application’s pages are placed in local memory. For the vast majority of production workloads at Meta, this is sufficient to guarantee that the performance of a workload using tiered memory is at a target level. We call it the fair share of local memory. There are two benefits of the approach. First, the mixture of local and CXL memory usage of an application is the common way in the literature to predict and control the performance levels of applications running on tiered memory systems [77, 145]. Secondly, the other potentially contended resource—CXL memory bandwidth—is also tied to the ratio of local vs. CXL memory usage; controlling the ratio is sufficient to also manage the contention on CXL bandwidth for capacity-bound workloads.

However, determining the fair share of an application is challenging for the following reasons. First, naively setting the fair share according to the system-wide local memory

vs. total capacity ratio, or evenly dividing the local memory capacity, leads to suboptimal performance as applications' varying memory access patterns are not accounted for. Secondly, the application owners at Meta have reported that it is difficult to determine the fair share as there is not a standard procedure and toolkit for that. Thirdly, the job scheduler may frequently decide to colocate a new set of jobs on a server that has not appeared before; different job combinations demand different memory fair shares.

5.4.2 System Overview

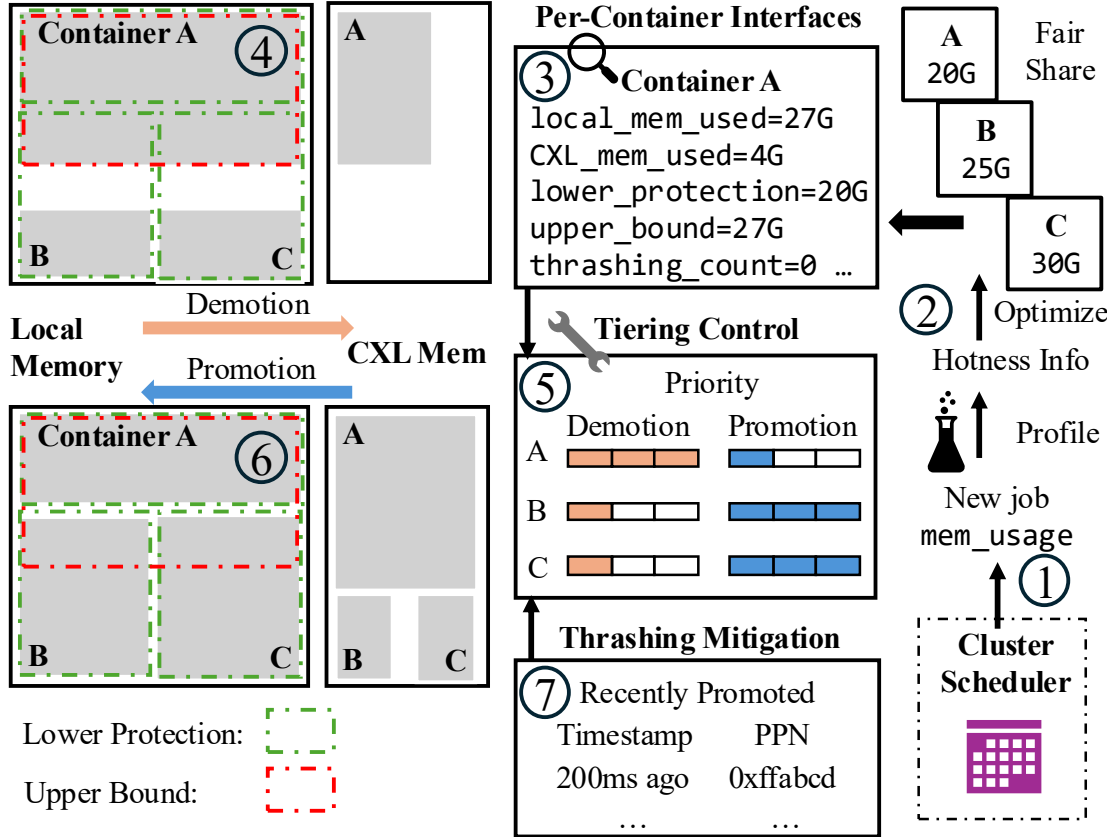


Figure 5.4: Equilibria overview.

Equilibria is a holistic solution for multi-tenancy on tiered memory systems. It has two main components: a tiering daemon in the userspace for managing jobs, and a OS kernel that enforces the memory fair share. The userspace daemon communicates with the cluster job scheduler for job information, automatically determines the local memory fair share of each colocated workload on the server, and applies the fair share values to the job containers. The OS kernel regulates the priorities for demotion and promotion for containers according to their current memory usage across memory tiers and their local memory fair share, enforcing the fair share. The two components work together to satisfy performance goals for colocated workloads in multi-tenant data centers with minimal per-job configuration.

Figure 5.4 gives a high-level overview of Equilibria and how it ensures memory placement fairness under multitenancy. When the cluster scheduler starts a new job on a host, it notifies Equilibria with the declared memory usage of the job (Step ①). Once the new job has gone through initialization and is in a stable state, Equilibria profiles its physical memory usage and identifies the coldness and expected memory bandwidth usage of it in small chunks. The profiling data, along with that of all other colocated workloads, is passed to an optimizer that maximizes memory offloading while respecting CXL memory capacity and bandwidth constraints (Step ②). The optimizer determines the local memory fair share for each job on the host, and applies it as lower protection (green dotted boxes), the guaranteed local memory usage, to job containers via per-container interfaces (Step ③). The job spec of A additionally sets an upper bound of its local memory usage (red dotted box). Equilibria also exposes the local and CXL memory usage and tiering activities, such as demotions, promotions, and reclamations, via the per-container interfaces, which is used for fleet performance monitoring. The OS component of Equilibria enforces the fair share. In the example, Container B and C are using less than their fair share, allowing Container A to opportunistically use take up the unused local memory. However, A is stopped from using all free local memory by its upper bound and has to spill 4 GB to CXL memory (Step ④). After a while, the memory demands of Container B and C increase, and the system local memory capacity is contended. Equilibria starts background demotion to free up local memory for hot new allocations. It prioritizes demotion and deprioritizes promotion for Container A since it is exceeding its local memory lower protection; conversely, Equilibria deprioritizes demotion for Container B and C so that they are guaranteed to get their local memory lower protections (Step ⑤). Soon, the local memory usage of containers converge to their fair share under contention (Step ⑥). Finally, Equilibria is always detecting instances when pages are frequently migrated between local and CXL memory, a sign that they are thrashing due to unfavorable access patterns and excessively large working sets, and slowing down promotions for thrashing containers to reduce noisy neighbor effects (Step ⑦).

5.4.3 Automatic Determination of Fair Share

Equilibria automatically determines whether a job can run on a host and what the local memory fair share of a set of colocated jobs should be, and configures the OS to enforce the fair share and maintain job SLO.

Admission Control

When the scheduler attempts to place jobs, it relies on the amount of free local and CXL memory reported by the tiering daemon to determine if a job fits on a host. To reduce the likelihood that a new job will create too much pressure in local memory that degrades the performance for every existing job on the host, Equilibria requires that a host has the amount of free local memory greater than 50% of the new job’s declared memory size to be eligible for receiving the new job. Other scheduling constraints such as CPU cores, memory capacity, disk, networking, etc. still apply. Once the scheduler selects a host to

run the new job, the decision will be communicated to the tiering daemon once the job successfully starts on the host.

Profiling Page Hotness

Equilibria first waits for the job to initialize and reach a stable running state. The tiering daemon in Equilibria then profiles the container for ten minutes to determine its memory access patterns and page hotness. The profiling is by the event-sampling of L3 load misses that correspond to memory fetches [11, 93, 145], which are the main contributor to performance degradation from using CXL memory. As we will show in the evaluation, the CPU overhead of the profiling is low.

Equilibria profiles the physical memory that is owned by the container in chunks of 2 MB, and tracks accesses to the memory chunks. Chunk size is determined by a balance of fine-grained information and profiling overhead. From profiling Equilibria produces the following information for each chunk i of memory: the coldness of the chunk W_i that is the length of time it has not seen accesses, and B_i that is the expected memory bandwidth incurred by the chunk. Specifically, B_i is calculated from total bandwidth BW , the number of samples in the chunk S_i , and the total number of samples obtained S , as $B_i = BW * S_i / S$.

Solving the Optimization Problem

Using the profiling data, Equilibria performs an optimization pass to determine an optimal distribution of local memory fair share across colocated workloads on the host. It pre-processes the profiling data of all colocated workloads on the host by creating a union set of all their physical memory chunks (i, B_i, W_i) . To prevent placing hot pages in the CXL memory and unduly degrade workload performance, it sets W_i (akin to the weight of a chunk) to zero if it is hot. Equilibria considers chunks that have been not accessed for more than two minutes as *cold*; this is the same threshold used in prior work [145] characterizing cold pages in data centers. Then, Equilibria solves the following constrained optimization problem:

Objective: Maximize $\sum_i X_i W_i$, where $X_i \in \{0, 1\}$ indicates whether a chunk is selected to be in the CXL memory.

Constraints:

1. **Capacity:** $c \sum_i X_i \leq Z$, where c is chunk size and Z is the capacity of CXL memory.
2. **Bandwidth:** $\sum_i X_i B_i \leq b$, where b is the knee of the bandwidth-util curve of CXL memory.

In essence, Equilibria jointly optimizes for the amount of memory offloaded to CXL and the coldness of the pages offloaded, while respecting the CXL memory capacity and bandwidth limits. To realistically solve the integer programming problem, we relax it to a linear programming problem. Specifically, we relax X_i be continuous in $[0, 1]$, and count a fractional X_i as partly in the CXL tier. Equilibria then deploys a linear programming solver [66] to solve the problem in polynomial time. Finally, Equilibria calculates the optimal memory amount of the new job in the CXL tier as cX_i , and subtract it from the declared memory usage of the job to get its local memory fair share.

5.4.4 Container-level Observability and Control

Equilibria introduces per-container tiering observability and controls using the cgroup [125] interfaces. It extends the tracking of per-container memory usage and management operations to a new dimension: memory tier, which supports fleet-wide performance monitoring, debugging and analysis. Equilibria adds within the cgroup’s data structure new memory usage counters per tier, and tiering activity counters, which cover successful and attempted demotions and promotions. When a page is allocated or freed, the tier of the page is identified, and the relevant usage counters of the owning container are updated. Similarly, when a page is migrated, the relevant promotion or demotion counters of the container are updated.

Equilibria adds two parameters as the interface for specifying the per-container local memory fair share. The first parameter, *lower protection*, is the amount of local memory protected from demotion or reclaim due to competing memory usage by other colocated containers when local memory is fully utilized. The second, *upper bound*, is the maximum amount of local memory a container can use before demotion reduces its local memory usage. They serve different purposes. The lower protection guarantees that a workload will get its fair share of memory regardless of the behavior of its neighbors, and avoids SLO violations. It still allows a container to opportunistically use available local memory exceeding its fair share, making local memory allocation work-conserving and enabling burstable performance. The upper bound, on the other hand, is a hard stop of a container’s local memory usage, useful when such higher than usual performance is undesirable for capacity planning, or when free local memory should be reserved for new workloads to quickly ramp up. These parameters enables the balancing of fairness, performance, and predictability in tiered memory systems. The automatically determined local memory fair share in Section 5.4.3 is applied as lower protection, because by default we want to be work-conserving and allow opportunistic use of extra local memory for higher host throughput. Each job also has the opportunity to override the values and set the upper bound if necessary.

5.4.5 Modulating Demotions

Equilibria manages the rate of demotion for each container to enforce fair share. When there is free local memory, containers are allowed to use more local memory than their lower protection. This approach improves the utilization of local memory, which offers lower latency compared to CXL memory, thereby optimizing the server’s total throughput.

When local memory is fully utilized, Equilibria begins to assign priorities to containers for demotion. Containers with local memory usage lower than their lower protection are exempted from demotions. Those exceeding their lower protection are targeted for demotions with intensity proportional to the overage. More specifically, if n_{cgroup} , $n_{protection}$, n_{lr_u} are the container’s local memory usage, lower protection and the size of a local memory LRU list of the container, respectively, the number of pages from the LRU list to be

considered for demotion d_{scan} is calculated as:

$$d_{scan} = \frac{n_{lru}(n_{cgroup} - n_{protection})}{n_{cgroup}} \quad (5.1)$$

After d_{scan} is calculated for a container, it is passed to the reclaim algorithm in Linux. Finally, a set of least recently used pages owned by the container will be selected and demoted. The process is repeated for every container in the system to free up local memory. Compared to Linux, Equilibria differentiates tiers of memory and accepts different memory lower protection and upper bounds for each tier in the hierarchy, whereas Linux does not consider memory from different tiers as needing separate resource control due to their distinct access performance. Over the long term, the prioritization of demotion by Equilibria applies higher demotion pressure to containers that use more local memory than their fair share given their neighbor containers, converging to a stable state that satisfies the fair share of all containers.

Equilibria enforces the upper bound by monitoring a container’s memory usage whenever it allocates memory. Unlike the lower protection, a container cannot use more local memory than the amount specified in upper bound. As the usage approaches the upper bound, Equilibria initiates a kernel background thread to demote pages and decrease the container’s local memory footprint, gently enforcing the limit with minimal disruption to the application’s execution. If the container’s local memory usage reaches or surpasses the limit during allocation, Equilibria requires the allocating thread to synchronously perform demotion before resuming userspace execution. This approach slows down future allocation requests for applications that try to rapidly allocate local memory without resorting to out-of-memory kills.

5.4.6 Regulating Promotion

Equilibria de-prioritizes promotion of containers that use more than their fair share of local memory, in concert with the prioritization of demotion for such containers to transition their memory placement to the desired state.

Equilibria categorizes a container as “promotion throttled” in either of the two cases: if its local memory usage exceeds its lower protection and the local memory is fully utilized, or if its local memory usage is approaching or exceeds its local memory upper bound. During scanning of promotion candidate pages for a container marked as promotion throttled, Equilibria reduces the number of pages to scan via the following calculation, where p_{scan} is the size of the container’s CXL memory footprint to consider for promotion, and p_{base} is the unthrottled promotion scanning rate:

$$p_{scan} = p_{base} * \max\left(\frac{n_{protection}^4}{n_{cgroup}^4}, \frac{1}{16}\right) \quad (5.2)$$

In other words, promotion scanning is slowed down in proportion to the fourth power of the overage amount over a container’s guaranteed share of local memory. We choose the power function so that the throttling starts minor but intensifies rapidly with more overage, i.e.,

promotion scan is at 96% of the normal size when the overage is 1%, but at 10% overage promotion scanning is only 68% of the base rate. The reason behind this is that a small overage is likely due to a burst of allocations that cause a temporary mismatch between the speed of demotion and allocation, which will subside on its own; on the other hand, a larger overage signals the inability of the demotion to keep up, necessitating a more drastic slowdown of promotion to more effectively pare back the container’s local memory usage. Notice that Equilibria still allows promotions to proceed when the local memory usage of a container exceeds its fair share because it may have very hot pages in CXL memory; otherwise, those pages may be stranded in CXL memory and excessively slow down its execution. By allowing a minimum of one-sixteenth of the base promotion scanning rate, Equilibria trades memory placement convergence speed for application performance. Such regulating of promotion ensures that demotion and promotion are not fighting each other.

5.4.7 Mitigating Thrashing

Thrashing is when pages are repeatedly and rapidly promoted and demoted between local and CXL memory. It can occur when a workload’s hot working set surpasses the local memory capacity, leading to pages being frequently promoted to local memory only to be demoted shortly thereafter to accommodate other hot pages, creating a cycle.

Thrashing between memory tiers is detrimental to performance, diverting compute cycles and memory bandwidth from application execution. In a CXL tiered memory system, it is particularly undesirable, as direct access to CXL (albeit at higher latency) is more efficient than migrating pages to local memory only to demote them again before they are subsequently accessed. Furthermore, thrashing increases both CXL and local bandwidth loads, thereby increasing access latency for direct CXL accesses as well. For this reason, in multi-tenant environments a thrashing container can create noisy neighbor problems, increasing the access latency for neighbors and impacting their performance. Therefore, mitigating thrashing is an essential component of Equilibria.

Equilibria identifies thrashing containers and limits their promotion rates. Equilibria maintains lightweight tracking of recently promoted pages by sampling them in a fixed-size hash table that records the physical page number and timestamp right after a promotion. During demotion, if the page is found in the hash table and the elapsed time between promotion and demotion is lower than a threshold of 30 seconds, the event is considered a thrashing occurrence. The container that owns the page will have its thrashing counter incremented. Equilibria includes a background thread that wakes up every 5 minutes and performs the following task: first, it clears the hash table that tracks recently promoted pages, moving the thrashing detection to the next period. Second, it calculates the rate of change in the thrashing counter of each container over the last period. If the rate exceeds a threshold 1000 pages per second, the promotion rate for pages owned by the container will be halved for the next period. And if a previously thrashing container no longer does so, its promotion rate will be doubled for the next period until the normal promotion rate is restored.

5.4.8 Implementation

We implemented the OS kernel on top of the Linux 6.11 using approximately 1,000 lines of code, and the userspace tiering daemon in 2000 lines of C++ code that relies on OR-Tools [66] as the solver. Equilibria integrates with the job scheduler to receive the proposed new job to be launched, determines the new fair local memory share of all the colocated workloads on the server, and applies the new fair share values to containers. Equilibria then works in the OS to track and export tier usage and tiering operation counts for containers, enforce fair share, and mitigate thrashing.

The kernel changes are implemented in a modular way with upstreaming to Linux in mind, and they are contained in the cgroups, demotion, and promotion code of Linux, making it easier to test and upstream. By adhering to the existing cgroup-based memory resource control and pursuing an upstream-first approach, we avoid ongoing code maintenance costs associated with designs that heavily deviate from how Linux currently manages memory. We have shared the OS patches with the Linux community and a subset of patches have been accepted upstream.

5.5 Evaluation

5.5.1 Methodology

We performed experiments on two types of Meta servers equipped with CXL memory, denoted as small and large servers. Both are single-socket servers with CPUs of the AMD Turin generation. The large server features a local to CXL memory ratio of 3:1, including 768GB of local DDR5 memory and 256GB of DDR4 DRAM as CXL memory. It has an unloaded CXL memory access latency of 252ns. The small server has 256GB of local DDR5 memory and 64GB of DDR4 DRAM as CXL memory. The CXL memory expanders used in the evaluation are CXL 2.0 Type 3 devices, reusing previously-decommissioned DDR4 memory.

We compare Equilibria against the upstreamed memory tiering support in Linux kernel, which is derived from TPP [145]. The servers have sufficient combined local and CXL memory capacity for colocated workloads, and swapping is disabled. Equilibria automatically determines the local memory fair share of colocated workloads (§5.4.3) in the evaluation. Although we do not provide a quantitative evaluation of the container-level observability of memory tiering provided by Equilibria, most of the evaluation and motivation studies in this work are only possible because of it.

We evaluate three of Meta’s production workloads: Cache, Web, and Continuous Integration (CI), as well as two benchmarks from DCPperf [174]: SparkBench and TaoBench. *Cache* is a distributed object cache that provides low latency access, defined consistency models, and common query semantics. It is the foundation of many internal and user-facing workloads at Meta. Cache receives replayed requests from production traces. *Web* is an application server that hosts many of Meta’s user-facing services. *CI* automates the process of building and testing code changes before integrating them into a shared repository. Both Web and CI serve live requests in production. *DCPerf* is a suite of data center

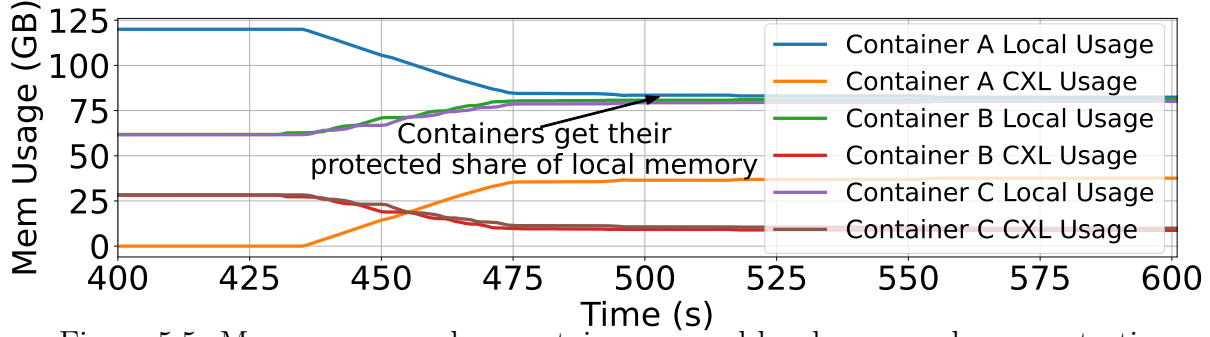


Figure 5.5: Memory usage when containers exceed local memory lower protection.

benchmarks that closely approximates major services in data centers.

5.5.2 Validating Tiering Control

In this section, we validate Equilibria’s control of tiering through page migration management and thrashing mitigation. These experiments were conducted on a small server. We designed a microbenchmark that allocate a predetermined amount of memory and access it through sequential passes. To simulate a multi-tenant environment, we launched three instances of this workload concurrently, each running within its own container. We used microbenchmarks due to their deterministic behavior and ease of analysis, allowing for clear reasoning about the system’s behavior. We consider the following four conditions: whether there is local memory pressure, whether there is unused local memory protection that can be donated, whether the upper bound is set, and whether there is thrashing. In the following experiments, the local memory lower protection is manually set to 80GB. No limits are imposed on CXL memory usage for any containers.

Local memory is preferred

When local memory is not contended, each container is allowed to reside entirely in local memory, even if exceeding its lower protection value (80GB). To validate this behavior, we launched three microbenchmarks with respective footprints of 120GB, 40GB, and 40GB. Given that their combined footprint (200GB) is less than the local memory capacity, no contention occurs. We observe that Equilibria allows all three containers to be fully resident in the local memory. This behavior of Equilibria increases the system-wide utilization of local memory to get potentially higher total throughput out of a server.

Lower protection is enforced

In this scenario, we launch three instances of the microbenchmark having memory footprints of 120GB, 90GB, and 90GB, respectively, each of which exceeds the per-container local memory lower protection. When the local memory is fully utilized and contended, each container’s local memory footprint eventually converges to its lower protection value of 80GB. The remaining memory footprint for each container is then spilled over into CXL

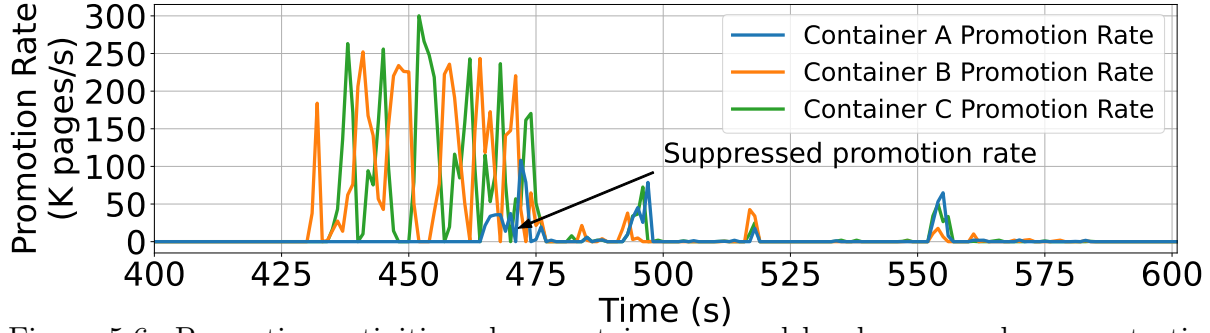


Figure 5.6: Promotion activities when containers exceed local memory lower protection. Container A’s promotion rate (blue line) is throttled.

memory, with Container A having 40GB and Container B and C having 10GB each, as illustrated in Figure 5.5.

We also observed that Container A is demoted more aggressively compared to the other two containers, with a rate of up to 250K pages per second. This is due to Container A exceeding its local memory lower protection by 50%, whereas Containers B and C only exceed it by 13%. As a result, Container A is demoted more heavily to bring its memory usage down to its fair share. Additionally, we note that the promotion rate of Container A is suppressed from time 425s to 475s, as shown in Figure 5.6. Despite using more CXL memory and having more hot candidate pages in CXL for promotion, Container A’s promotion rate is intentionally limited by Equilibria. This ensures that promotions do not interfere with converging to a steady state.

Unused lower protection is donated

We launch three microbenchmark instances with memory footprints of 120GB, 70GB, and 70GB. Container A is launched first and utilizes all available local memory before the other containers start. The memory usage of three containers converges over time. Container A opportunistically uses 20GB more local memory than its lower protection, even when local memory is contended.

This is because Container B and C donate their unused local memory share to Container A. In this case, where donation happens, Equilibria still ensures that B and C have all their protected local memory share and fully reside in local memory. This experiment demonstrates the work-conserving nature of Equilibria’s lower protection mechanism. In production, a similar scenario can help workloads achieve higher performance when the system has free local memory. Note that Containers B and C are completely protected from demotions because their memory usage falls under their local memory lower protection, and we find that only Container A experiences demotion activities. This means that colocated workloads do not need to worry about degraded performance when their unused share of local memory is donated.

Upper bound is enforced

In this experiment, three microbenchmark instances with memory footprints of 120GB, 40GB, and 40GB are used, similar to the “no local memory pressure” case. However, Container A’s local memory upper bound is set to 80GB. As a result, despite ample system-wide local memory availability, Container A is restricted from using more than 80GB of local memory, forcing it to spill 40GB into CXL memory. This scenario represents scenarios where minimizing performance variability is paramount, and local memory should be reserved for new neighbors rather than being opportunistically utilized by existing workloads.

Thrashing is mitigated

In this experiment, we demonstrate the effectiveness of Equilibria in detecting and mitigating thrashing workloads, resulting in improved system-wide performance. We tune the aforementioned microbenchmark to create a thrashing workload as follows: it allocates 100GB of memory and accesses it in blocks such that the memory is hot enough to trigger page promotions from CXL memory but not enough to re-access the promoted pages before they are demoted. The local memory upper bound for the container is set to 10GB. We run two other microbenchmark instances with 50GB of memory footprint each, representing normal workloads that do not thrash. These workloads fit within their local memory lower protection. We compare two scenarios: one in which thrashing mitigation is disabled and another in which it is enabled. Without thrashing mitigation, the thrashing workload experiences aggressive page migrations (over 100K pages per second), causing interference with the other two normal workloads, which access memory at a rate of 290K pages per second. However, when thrashing mitigation is enabled, Equilibria detects the thrashing behavior and reduces page migrations to thousands of pages per second, significantly decreasing system-wide memory management overhead. As a result, the colocated well-behaving programs can sustain access to memory at a rate of 310K pages per second, achieving a 7% performance improvement.

5.5.3 Characterizing Fair Share Determination

In this section, we characterize the automatic local memory fair share determination of Equilibria from the perspectives of time and CPU and memory overhead, and examine the mechanism in action using open-source benchmarks.

We select two workloads from DCPerf: SparkBench, which represents data analytics [233]; and TaoBench, which represents a caching service [36]. In our experiment, we stack three instances of TaoBench (Containers B, C, and D) and one instance of SparkBench (Container A) per server. TaoBench has a steady memory usage and access pattern, while SparkBench has bursty memory usage and varying access hotness over its data analytics phases. Since SparkBench completes faster than TaoBench, we run SparkBench in a loop.

After new jobs (Container A,B,C,D) start running on a host and have initialized, Equilibria spends 10 minutes profiling the workload, and spends another 3 seconds to solve

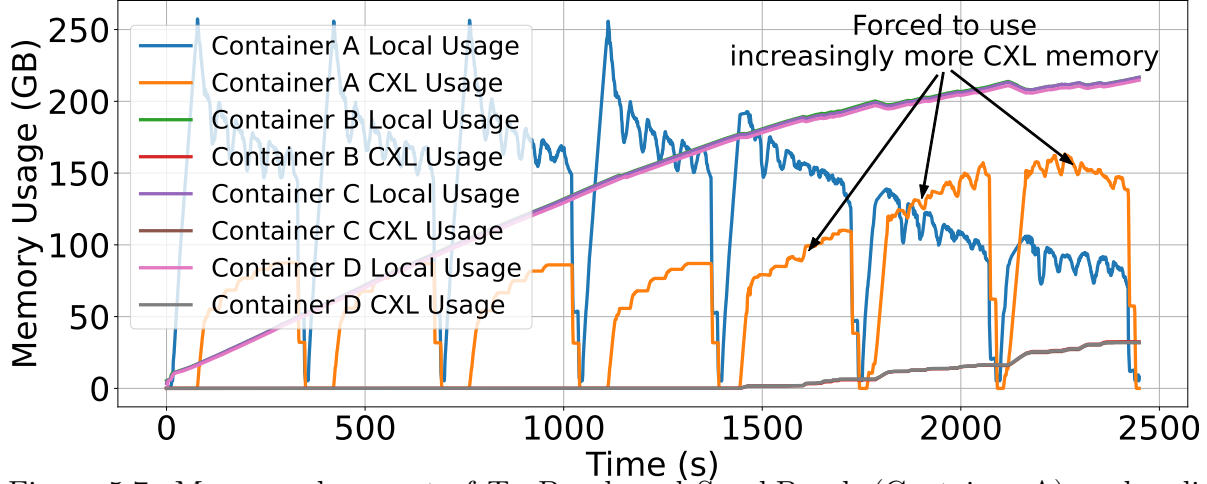


Figure 5.7: Memory placement of TaoBench and SparkBench (Container A) on baseline Linux. Container A is forced to use increasingly more CXL memory and less local memory.

the integer linear programming problem. The profiler and the optimizer consume, at their peak, 180 MB of memory and one CPU core. This shows that Equilibria can quickly determine the fair share of new jobs just minutes after they start, allowing the system to enforce local memory fair share rapidly and preserve application SLOs.

Looking deeper into the process, we see that Equilibria decided to place 136 GB of memory of SparkBench into CXL memory with an expected CXL bandwidth usage of 6 GB/s. Overall, the local memory fair share of the four jobs correspond to 238 GB of CXL memory usage, ensuring a well-utilized CXL tier, while keeping the CXL bandwidth utilization low and avoiding elevated access latency under bandwidth exhaustion.

5.5.4 Evaluating Equilibria with Benchmarks

Following the workload setup in §5.5.3, we study the memory placement on baseline Linux and Equilibria. Figure 5.7 shows the timeseries of memory placement across tiers of each job on baseline Linux. As TaoBench ramps up and consumes more memory, SparkBench with its less hot access patterns will be forced to use increasingly more CXL memory (orange line) and less local memory (blue line) in later iterations. This leads to drastically slower performance for SparkBench. We obtain on Equilibria 11.2 queries per hour (QPS) for SparkBench versus 4.2 QPS on Linux, which is a $1.7\times$ throughput improvement. The benefits come from the higher and more stable share of local memory that SparkBench is able to use when running under Equilibria as the maximum local memory per container is constrained. Looking into memory placement with Equilibria, TaoBench is forced to spill all subsequent memory allocations past the 192GB upper bound into CXL memory, preserving free local memory for the short-lived and bursty SparkBench to use. Notably, the local memory upper bound in Equilibria only marginally decreases SparkBench’s performance by less than 0.3%, demonstrating that applications have a wide range of sensitivity to memory placement and introducing fairness in Equilibria does not necessarily harm total

system throughput. This experiment demonstrates that Equilibria prevents the possible severe performance degradation when heterogeneous workloads are colocated.

5.5.5 Deploying Equilibria in Production

We evaluate Equilibria with three case studies of deploying production workloads at Meta, demonstrating how Equilibria meets workload SLOs and improves workload performance with flexible fairness policies. The three workloads are all memory-capacity-bound and have low CXL bandwidth utilization (<10% available bandwidth) in our experiments.

Cache

Cache exhibits random access patterns over its entire address space, with a significant portion of its memory footprint being hot (e.g., up to 60% of the cache may be accessed concurrently). We stack two instances of Cache (Container A, B) on large servers, which fully saturates the application-usable memory capacity of the server. Because the two instances are homogeneous, the SLO calls for nearly identical cache look up latency.

We first start both the containers at the same time on the baseline Linux without fairness support. With both containers running homogeneous workloads, their memory usage across the tiers was supposed to be homogeneous. However, with Linux, we see that for Container A, 90% of its memory stays in local memory with the rest spilling to CXL memory, whereas Container B only keeps 70% of its memory in local memory and uses significantly more CXL memory. This unfair memory split across the memory tiers results in uneven application-level performance. More local memory helps Container A to serve more cache queries from the fastest memory tier: throughout the run, A has $1.15\times$ better on average and as high as $3.3\times$ better P99 latency over B.

Although A is more advantageous in keeping an unfairly large amount of local memory when colocated with B under normal circumstances, without fairness support it can still fall victim to noisy neighbor effects. To demonstrate this, we stop the loads to B, then artificially inject burst loads to B so that its memory utilization jumps from 0% to 90% within a minute. On Linux, rapid allocations by B forces A to demote 15% of its anon memory to CXL tier and causes instability in A’s performance. Cache being a latency sensitive workload, A experiences a huge spike in its tail latency and experiences a high failure rate – request failures spike from 340 per minute to 83K per minute). As a result, A’s throughput drops by 65%.

We then run Cache on Equilibria, and we set the local memory lower protection to 70% and upper bound to 75% of the container’s memory usage. Equilibria is able to enforce the fair share of memory for the two containers. In stable state, Container A and B maintain almost an identical memory usage across the tiers – on average, 72% and 73% of their memory footprint is on local memory, respectively. With a more balanced local memory share, B’s average P99 latency reduces by 47% compared to Linux. Meanwhile, A’s latency also reduces by 52% because of a 97% reduction in NUMA hint faults – setting the upper bound for both containers has limited the contention of local memory and every container spends less time on demotion and promotion that inflate application tail latency.

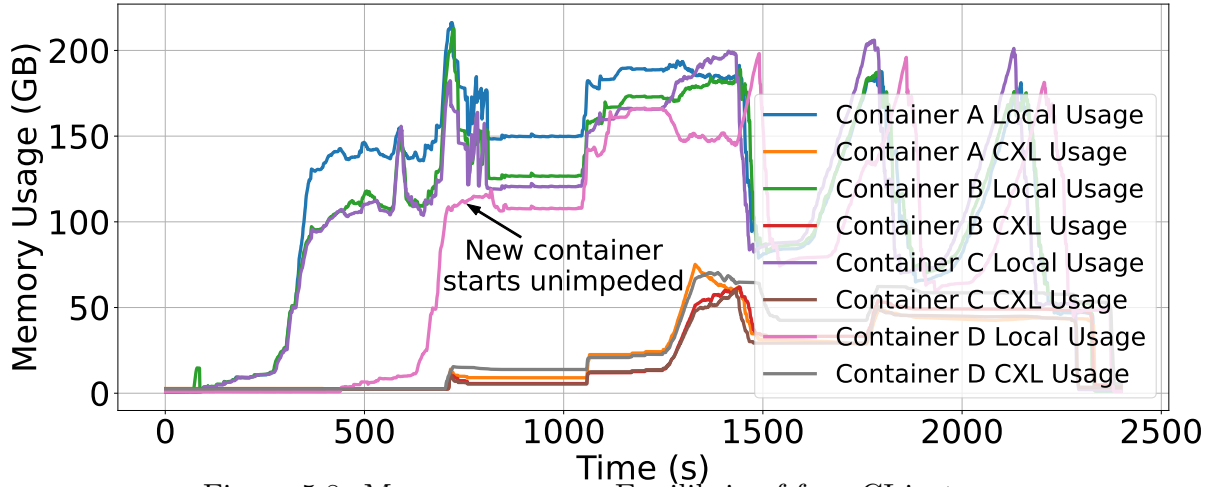


Figure 5.8: Memory usage on Equilibria of four CI instances.

The result is that B has only 7% higher average P99 latency over A, and for Cache, this uniform latency is key to meeting its SLO as an uneven latency in the object cache can propagate to other dependent services and cause cascading violations of performance guarantees in user-facing services. This experiment shows that Equilibria helps colocated homogeneous workloads meet latency SLOs and avoid noisy neighbor effects.

CI

We evaluate colocated CI instances on large servers. We stack four containers to execute four build jobs simultaneously, and compare runs on Equilibria and baseline Linux. As the service is not customer-facing, it does not have a stringent SLO. Instead, the primary goal of Equilibria is to ensure that no containers are unfairly penalized due to local memory contention and to optimize for total job throughput. We set on Equilibria the local memory lower protection to 192GB per container. This represents a simple fairness policy that does not need in-depth profiling of the workloads. CI has spiky memory capacity usage as depicted in Figure 5.8 showing its memory usage running on Equilibria, because linking phases of build jobs tend to be more memory-intensive than other phases. In the absence of local memory pressure, Container A, B, C ramp up to freely utilize the available local memory capacity exceeding their fair share, leading to higher performance than stricter policies. Then, shown in the figure from time 500s to 800s, Container D starts and rapidly allocates local memory, while other containers are forced to reduce their local memory footprint as they exceed their protected share. We see that on Equilibria, Container D starts unimpeded and can put over 90% of its memory footprint in local memory, whereas on Linux it is forced to put more than half of its footprint in CXL memory. As for performance, Equilibria achieved a 3.3% drop in average build time across the four build jobs relative to the Linux baseline. This experiment shows that with Equilibria, a simple policy derived from system memory capacity ratio is sufficient to improve total system throughput for some workloads in multi-tenant systems with CXL memory.

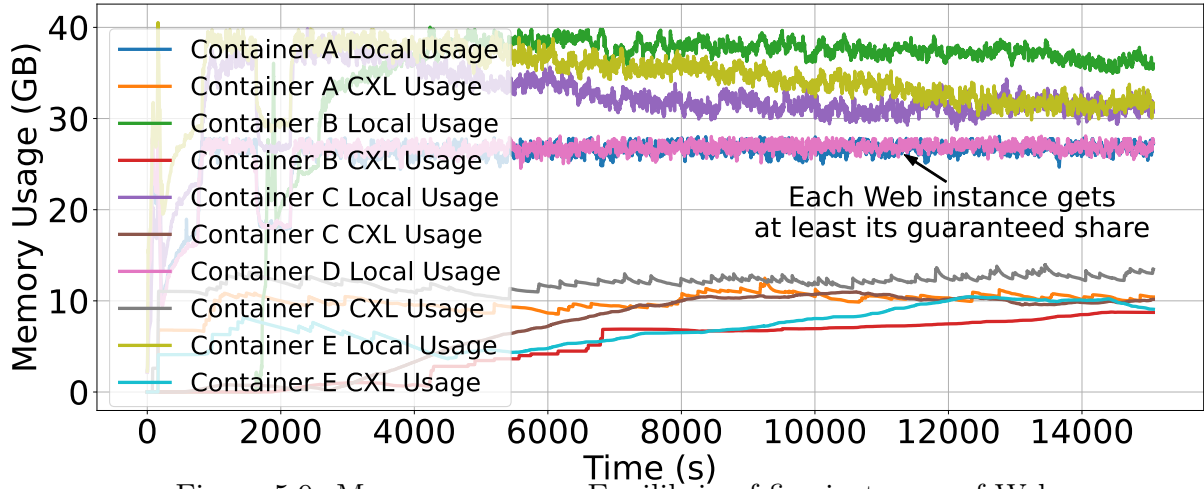


Figure 5.9: Memory usage on Equilibria of five instances of Web.

Web

We run colocated Web instances on small servers. At runtime, an instance is specialized to serve a subset of traffic (partition) using just-in-time compilation. Therefore, different instances have different memory usage and access patterns despite sharing the same program binary. Web is not memory capacity-bound on a per-instance level (i.e., increasing an instance’s memory usage will not yield improved performance), and the gains of using CXL memory come from having more instances per server. We consider three cases: stacked Web instances on Equilibria, on Linux baseline, and a single instance running alone using only local memory. The performance of Web is measured as its average throughput per instance measured in requests per second (RPS) when internal latency limits are satisfied, and Web requires an SLO of less than 2% slowdown compared to running alone using only local memory. In the two stacked cases, we stack five instances of Web per server, with three instances belonging to partition A (Container B, C, E) and two instances belonging to partition B (Container A, D). We set on Equilibria the local memory lower protection to 28GB per container based on a profiling of the hot memory footprint of Web over a 10-minute period, ensuring there is sufficient local memory for an instance of Web to allocate its hot footprint without overcommitting the local memory. The policy still allows some instances to utilize spare local memory capacity.

First, we compare end-to-end performance. On Equilibria, stacked Web instances do not lose performance compared to the single instance case, achieving 36 RPS for partition A and 80 RPS for partition B, meeting the service SLO. On baseline Linux, although stacked instances serving partition A do not lose performance, instances serving partition B get only 74 RPS, missing the $<2\%$ performance loss SLO. In other words, Equilibria allows 8% more throughput for Web partition B than baseline running with multi-tenancy and CXL memory.

Next, we analyze the memory usage of colocated Web instances on baseline Linux. For Container A and D, we see their local memory usage trending downwards over time, decreasing by over 5GB over the course of the experiment. This is because on Linux, their pages manifest as less hot than pages from other colocated Web instances and are

gradually migrated to CXL memory. The loss of local memory share for them leads to lower performance compared to Equilibria.

Contrast this with memory usage on Equilibria in Figure 5.9. Local memory lower protection is enforced, as every container is able to use at least 28GB of local memory. Note that Container A and D are using exactly 28GB of local memory at stable state, showing that Equilibria prevents instances on partition B from being forced to use an excessive amount of CXL memory as on Linux. The experiments demonstrate that Equilibria can enforce the memory fair share policy for colocated containers and enable colocated performance that is close to running alone in systems with CXL memory.

5.6 Related Work and Discussions

Prior work [2, 16, 37, 55, 83, 110, 121, 123, 127, 145, 171, 173, 184, 196, 207, 212, 223, 226, 230, 238, 240] transparently manages memory placement of applications in the OS for systems with tiered memory. In terms of design objectives, prior work has primarily focused on optimizing memory placement for the entire system. However, fairness in multi-tenancy is important as colocated workloads require a certain level of performance predictability and guarantees regardless of their neighboring applications’ behaviors. When prior work does address multi-tenancy, it relies heavily on coarse-grained workload categories, such as *critical/best-effort* [55, 123]. This ignores the problem that colocated “critical” workloads must each meet their performance SLOs but still suffer from noisy neighbor effects. In fact, at Meta, the vast majority of multi-tenancy setups are multiple instances of the same “critical” workload. Additionally, most prior work, except for TPP [145], is not upstreamed to Linux, making their production deployment difficult and long-term maintainability uncertain.

TPP [145] identifies cold pages to demote to CXL memory by leveraging Linux’s page reclaim algorithm, which already tracks page hotness. For promotion, TPP utilizes Linux’s NUMA balancing mechanism to identify hot pages in CXL memory. TPP, however, lacks any notion of fairness and is not designed for multi-tenancy. Another tiering work, MEMTIS [121], controls the ratio of the fast tier to capacity tier memory for its evaluation using the cgroup interface, supporting only an upper limit of the amount of fast tier memory that a cgroup can use, akin to allocating fast tier capacity to cgroups. Equilibria, in addition to supporting such upper limits, supports a much more powerful lower protection of the fast tier memory usage that is work-conserving and applied in relation to the local memory share of other colocated cgroups in the systems. Furthermore, demotions and promotions in Equilibria are regulated to conform to the lower protection, in contrast to the allocation-only enforcement in MEMTIS. Equilibria is designed and evaluated for multi-tenancy, while the cgroup limits in MEMTIS are only an aid to its evaluation.

Prior work in memory tiering has also used optimization techniques. Tierscape [116] calculates an optimal placement of a single application’s memory across compressed memory tiers based on tier access latency and frequency, but does not support colocated jobs in multi-tenant data centers. CFM [8] minimizes local *memory-time* product and rebalances local memory usage of colocated batch-processing jobs, but does not support latency-

sensitive production workloads. Equilibria, on the other hand, uses optimization problems for determining the memory placement of colocated production workloads while meeting their SLOs.

5.7 Conclusion

Equilibria is the first multi-tenant fairness solution for CXL memory tiering in data centers. It provides per-container observability, simple yet powerful mechanisms for flexible fairness policies, automatic memory fair share determination, and thrashing mitigation. Equilibria meets the performance SLOs of colocated workloads using CXL memory where TPP falls short, improves end-to-end performance of benchmarks and production workloads by up to $1.7\times$ and 52%, respectively, and is being deployed at Meta and upstreamed.

Chapter 6

Conclusion

Memory is one of the oldest abstractions and mechanisms in computers, yet it has recently come under strain due to the rapid expansion of memory capacity and the proliferation of memory-intensive applications. The answer is to pursue memory efficiency by identifying bottlenecks and resolving them through the co-design of operating systems and computer architecture. This thesis approached the challenge from multiple complementary dimensions: access performance, hardware costs, and power consumption.

To improve memory access performance and reduce address translation overhead, this thesis presents two solutions that tackle the challenge from two angles. Contiguitas identifies unmovable allocations as the primary culprit behind the severe lack of physical memory contiguity in data centers. By grouping unmovable allocations in the OS and introducing hardware extensions to migrate previously unmovable pages used for device I/O, Contiguitas generates abundant physical memory contiguity. This newly found contiguity allows for the allocation of larger huge pages, yielding significant performance improvements for production workloads. Second, building on a sufficient supply of contiguity, LVM rethinks the rigid structure of radix page tables. By leveraging the regularity of application virtual address spaces, LVM builds learned indexes for address translation. This design drastically reduces the page walk overhead over state-of-the-art page tables and achieves substantial speedups in application execution.

To reduce the cost and power consumption of memory, this thesis presents two solutions that make the deployment of cheaper heterogeneous memory practical in data centers. Pensieve introduces a three-tier memory system that dynamically manages pages across DRAM, CXL memory, and SSDs. By employing a unified feedback-driven controller, Pensieve overcomes the capacity limitations of CXL memory and the performance limitations of SSD swap, safely offloading a significant fraction of memory while keeping application performance degradation minimal. Finally, Equilibria addresses the challenges of deploying tiered memory in multi-tenant environments. By ensuring fair memory sharing across memory tiers and mitigating noisy-neighbor interference through automatic fair-share determination and flexible placement policies, Equilibria consistently helps workloads meet their service-level objectives (SLOs) while avoiding performance interference.

Together, these works demonstrate that memory efficiency in modern data centers can indeed be drastically improved. By addressing the memory bottleneck from multiple an-

gles, this thesis provides the necessary foundations for the continued sustainable scaling of memory capacity in the next generation of data center computing. Importantly, these works have already made their way into upstream Linux and deployed in production datacenter across millions of servers.

6.1 Future Work

This thesis builds a solid foundation for efficient memory systems in data centers, but is only the first step in the quest for memory efficiency. Several avenues remain open for future exploration.

Flexible Page Table Structures Contiguity provides abundant physical memory contiguity, enabling a new set of page table designs that rely on this contiguity, which LVM leverages to make virtual memory more efficient. We are working on a new page table design, *Accordion*, that enhances radix-based translation by dynamically harnessing variable-sized contiguous physical memory during page table allocation. The design allows each page table level to cover significantly more entries of the lower levels, particularly at leaves, expanding their effective coverage and boosting locality. It requires minimal changes in hardware and the OS, and expands the coverage of page walk cache entries to reduce address translation overhead.

Application and Hardware Support for Memory Tiering In the context of tiered memory management, future work can explore leveraging application hints to make more intelligent placement decisions, such as guiding direct demotions from DRAM to specific lower tiers. Finally, building on the software-based multi-tenant fairness provided by Equilibria, we also envision the development of targeted hardware support for page migration and hotness tracking. Such hardware-software co-design would allow the OS to manage CXL and other tiered memory technologies with even greater precision and lower overhead.

Learned Index for Hardware Caches LVM opens new possibilities for applying learned indexes beyond page tables. We envision extending lightweight learned indexes to other hardware caching structures indexed by virtual addresses, such as TLBs, branch target buffers, and L1 caches. This helps reduce the conflict misses that increase miss rates in those hardware caches by tapping into the regularity of applications' virtual address spaces. Furthermore, applying this approach to physically indexed structures like last-level caches could mitigate hash-table-like collisions that degrade hit rates, but may require a more restricted application physical memory layout. To apply learned indexes to hardware caches, new software-hardware interfaces to manage them must be devised.

Bibliography

- [1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, 2006. 2.1, 3.1
- [2] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344654. doi: 10.1145/3037697.3037706. URL <https://doi.org/10.1145/3037697.3037706>. 4.1, 4.7, 5.1, 5.6
- [3] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting Hardware-assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*, Portland, Oregon, June 2012. 2.1, 3.1
- [4] Hana Alam, Tianhao Zheng, Mattan Erez, and Yoav Etsion. Do it yourself virtual memory translation. In *the Proceedings of ISCA*, pages 1–12, Toronto, Canada, June 2017. doi: 10.1145/3079856.3080209. 2.1
- [5] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*, 2017. 3.1
- [6] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, 2005. 3.6.2
- [7] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA '20)*, 2020. 2.1, 3.1
- [8] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for

- Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387522. URL <https://doi.org/10.1145/3342195.3387522>. 5.6
- [9] AMD. AMD I/O Virtualization Technology (IOMMU) Specification. <https://www.amd.com/en/support/tech-docs/amd-io-virtualization-technology-iommu-specification>, 2022. 2.2.1, 2.3.3
 - [10] AMD. Invalidate TLB Entry(s) with Broadcast. <https://www.amd.com/system/files/TechDocs/24594.pdf>, 2022. 2.2.1
 - [11] AMD. Amd research instruction based sampling toolkit, January 2025. URL https://github.com/jlgreathouse/AMD_IBS_Toolkit. 5.3.5, 5.4.3
 - [12] Nadav Amit. Optimizing the tlb shutdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 27–39, USA, 2017. USENIX Association. ISBN 9781931971386. 2.2.1, 2.7
 - [13] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down tlb shutdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387518. URL <https://doi.org/10.1145/3342195.3387518>. 2.2.1, 2.7
 - [14] Kiam Heong Ang, G. Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13(4):559–576, 2005. doi: 10.1109/TCST.2005.847331. 4.4.4, 4.7
 - [15] ArangoDB. Transparent Huge Pages Warning. <https://github.com/arangodb/arangodb/blob/2b84348b7789893878ebd0c8b552dc20416c98f0/lib/ApplicationFeatures/EnvironmentFeature.cpp>, 2018. 2.1
 - [16] Moiz Arif, Avinash Maurya, M Mustafa Rafique, Dimitrios S Nikolopoulos, and Ali R Butt. Application-attuned memory management for containerized hpc workflows. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 114–127, San Francisco, CA, USA, 2024. IEEE, IEEE Computer Society. 5.1, 5.6
 - [17] ARM. Arm confidential compute architecture, 2023. <https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture>. 1, 2.1
 - [18] ARM. TLBI (A64). <https://developer.arm.com/documentation/dui0801/1/A64-General-Instructions/TLBI--A64->, 2023. 2.2.1
 - [19] Jean-Philippe Aumasson, Willi Meier, Raphael Phan, and Luca Henzen. *The Hash Function BLAKE*. Springer, 2014. 3.7.3
 - [20] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. Avoiding tlb shutdowns through self-invalidating tlb entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 273–287, 2017. doi: 10.1109/PACT.2017.38. 2.2.1, 2.7

- [21] Jens Axboe. fio - flexible i/o tester rev. 3.41, April 2026. URL https://fio.readthedocs.io/en/latest/fio_doc.html. 4.6.3
- [22] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization*, 14(2):14:1–14:25, June 2017. ISSN 1544-3566. 2.5.3, 3.7.4
- [23] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*, 2010. 2.1, 3.1, 3.2.2
- [24] Thomas W. Barr, Alan L. Cox, and Scott Rixner. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*, 2011. 2.1, 2.7, 3.1
- [25] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*, 2013. 2.1, 3.1, 3.8
- [26] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhan. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629579. URL <https://doi.org/10.1145/1629575.1629579>. 2.2.1, 2.7
- [27] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2017. URL <https://arxiv.org/abs/1508.03619>. 4.1
- [28] N. Beckmann, P. Tsai, and D. Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 538–550, Los Alamitos, CA, USA, 2015. IEEE Computer Society. 2.3.3
- [29] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005. 2.4, 3.6.1
- [30] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, 2008. 2.1, 3.1
- [31] Abhishek Bhattacharjee. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, 2013. 2.1, 3.1
- [32] Abhishek Bhattacharjee. Translation-Triggered Prefetching. In *Proceedings of the*

- 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017. 2.1, 3.1
- [33] Abhishek Bhattacharjee and Margaret Martonosi. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*, 2010. 2.1, 3.1
 - [34] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, 2011. 2.1, 2.7, 3.1
 - [35] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. Reconsidering Java performance analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM, 2025. doi: 10.1145/3669940.3707217. URL <https://doi.org/10.1145/3669940.3707217>. 3.3.1
 - [36] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 49–60, USA, 2013. USENIX Association. 5.5.3
 - [37] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 79–92, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446713. URL <https://doi.org/10.1145/3445814.3446713>. 3. 4.1, 5.6
 - [38] Travis Campbell. Transparent Huge Pages on Hadoop makes me sad. <https://www.ghostar.org/2015/02/transparent-huge-pages-on-hadoop-makes-me-sad>, 2015. 2.1
 - [39] Prakash Chauhan, Chris Petersen, Brian Morris, and Jerome Glisse. OCP Hyper-scale CXL Tiered Memory Expander Specification, Version 1.0, Base Specification Template v1.2. <https://www.opencompute.org/documents/hyperscale-tiered-memory-expander-specification-for-compute-express-link-cxl-1-pdf>, October 2023. 4.2.2, 4.5.2, 4.6
 - [40] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '20, page 293–304, New York, NY, USA, 2020. Association for Computing

- Machinery. ISBN 9781450370479. doi: 10.1145/3368826.3377922. URL <https://doi.org/10.1145/3368826.3377922>. 5.3.5
- [41] Albert Cho and Alexandros Daglis. Starnuma: Mitigating numa challenges with memory pooling. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 997–1012, 2024. doi: 10.1109/MICRO61859.2024.00077. 5.1
 - [42] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. Coaxial: A cxl-centric memory system for scalable servers. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2024. doi: 10.1109/SC41406.2024.00101. 5.1
 - [43] Jeongku Choi. Memory prices surge up to 90% from Q4 2025. <https://counterpointresearch.com/en/insights/Memory-Prices-Surge-Up-to-90-From-Q4-2025>, February 2026. 4.1
 - [44] Cloudera. Transparent Huge Pages Warning. https://www.cloudera.com/documentation/enterprise/latest/topics/cdh_admin_performance.html, 2021. 2.1
 - [45] Edward I. Cohen, Gary M. King, and James T. Brady. Storage hierarchies. *IBM Systems Journal*, 28(1):62–76, 1989. 4.7
 - [46] Compute Express Link Consortium. CXL 3.0 White Paper. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf, 2022. 1, 2.3.3, 4.2.2, 5.1
 - [47] Jonathan Corbet. Memory compaction. <https://lwn.net/Articles/368869/>, 2010. 2.2.1
 - [48] Guilherme Cox and Abhishek Bhattacharjee. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’17)*, 2017. 2.1, 2.7, 3.1
 - [49] Lixiao Cui, Yijing Luo, Yusen Li, Gang Wang, and Xiaoguang Liu. When learned indexes meet persistent memory: The analysis and the optimization. *IEEE Transactions on Knowledge and Data Engineering*, 36(12):9517–9531, 2024. doi: 10.1109/TKDE.2023.3342825. 3.8
 - [50] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’14)*, 2014. 2.1, 3.1
 - [51] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An introduction to the compute express link (cxl) interconnect. *ACM Comput. Surv.*, 56(11), July 2024. ISSN 0360-0300. doi: 10.1145/3669900. URL <https://doi.org/10.1145/3669900>. 4.2.2
 - [52] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex:

- an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020. 3.1, 3.2.3, 3.4.2
- [53] The Linux Kernel Documentation. Shared Virtual Addressing (SVA) with ENQCMD. <https://docs.kernel.org/x86/sva.html>, 2023. 2.3.3
 - [54] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI’99)*, 1999. 2.1, 3.1, 3.2.2
 - [55] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582031. URL <https://doi.org/10.1145/3582016.3582031>. 4.1, 4.4.3, 4.5.2, 4.6, 4.7, 5.1, 5.6
 - [56] Stephane Eranian and David Mosberger. The Linux/ia64 Project: Kernel Design and Status Update. Technical Report HPL-2000-85, HP Labs, 2000. 2.1
 - [57] Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020. ISSN 2150-8097. doi: 10.14778/3389133.3389135. URL <https://dl.acm.org/doi/10.14778/3389133.3389135>. 3.1, 3.2.3
 - [58] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. Towards a machine learning-assisted kernel with lake. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 846–861, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575697. URL <https://doi.org/10.1145/3575693.3575697>. 3.3.2
 - [59] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL <https://www.usenix.org/conference/nsdi18/presentation/firestone>.

2.2.5

- [60] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), 2004. 2.4, 3.6.2, 4.1
- [61] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. *Theory of Computing Systems*, 38(2):229–248, February 2005. 3.2.2
- [62] Jacob Fox. Looks like memory prices are set to keep increasing over 2026 before falling. <https://finance.yahoo.com/news/looks-memory-prices-set-keep-152000968.html>, December 2025. 4.1
- [63] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, Cambridge, United Kingdom, December 2014. 2.1, 3.1, 3.8
- [64] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*, 2016. 2.1, 3.1
- [65] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*, 2014. 2.1, 2.6, 3.8
- [66] Google. Or-tools, April 2026. URL <https://developers.google.com/optimization/>. 5.4.3, 5.4.8
- [67] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 41–50, New York, NY, USA, June 2008. Association for Computing Machinery. ISBN 9781605581347. doi: 10.1145/1375634.1375641. URL <https://doi.org/10.1145/1375634.1375641>. 2.7
- [68] Mel Gorman and Patrick Healy. Performance Characteristics of Explicit Superpage Support. In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*, 2010. 2.1, 2.6, 3.8
- [69] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. Linux Symposium, 2005. 3.7.3
- [70] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — A System Implementor's Tale. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*, 2005. 2.1, 3.2.2
- [71] Boris Grot, Damien Hardy, Pejman Lotfi-Kamran, Babak Falsafi, Chrysostomos Nicopoulos, and Yiannakis Sazeides. Optimizing data-center tco with scale-out processors. *IEEE Micro*, 32(5):52–63, 2012. doi: 10.1109/MM.2012.71. 5.2.2
- [72] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings*

- of the 9th USENIX Conference on File and Storage Technologies, FAST'11, page 20, USA, 2011. USENIX Association. ISBN 9781931971829. 4.7
- [73] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341936. doi: 10.1145/2934872.2934908. URL <https://doi.org/10.1145/2934872.2934908>. 2.2.5
 - [74] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM International Conference on Virtual Execution Environments (VEE'15)*, 2015. 2.1, 2.6, 3.8
 - [75] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting virtual memory with midgard. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 512–525, 2021. doi: 10.1109/ISCA52012.2021.00047. 2.1, 2.6
 - [76] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting virtual memory with midgard. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 512–525, 2021. doi: 10.1109/ISCA52012.2021.00047. 3.7, 3.7.5, 3.8
 - [77] 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. ISBN 9798400723599. doi: 10.1145/3779212.3790198. URL <https://doi.org/10.1145/3779212.3790198>. 4.7, 5.4.1
 - [78] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. The virtual block interface: A flexible alternative to the conventional virtual memory framework. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1050–1063, 2020. doi: 10.1109/ISCA45697.2020.00089. 3.1
 - [79] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, Williamsburg, VA, USA, March 2018. 2.1, 3.1, 3.8
 - [80] Red Hat. Red hat enterprise linux network performance tuning guide, 2015. https://access.redhat.com/sites/default/files/attachments/20150325_network_performance_tuning.pdf. 2.2.5
 - [81] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback*

control of computing systems. John Wiley & Sons, 2004. 4.7

- [82] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2011. ISBN 978-0-12-383872-8. URL <https://dl.acm.org/doi/book/10.5555/1999263>. 4.3.2
- [83] Junliang Hu, Zhisheng Hu, Chun-Feng Wu, and Ming-Chang Yang. Demeter: A scalable and elastic tiered memory solution for virtualized cloud via guest delegation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP '25*, page 169–185, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400718700. doi: 10.1145/3731569.3764801. URL <https://doi.org/10.1145/3731569.3764801>. 4.1, 5.6
- [84] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, 1993. 2.1, 3.1, 3.2.2
- [85] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL <https://www.usenix.org/conference/osdi21/presentation/hunter>. 1, 2.1, 2.6, 2.7, 3.1, 3.8
- [86] IBM. PowerPC[®] Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors. https://wiki.alcf.anl.gov/images/f/fb/PowerPC_-_Assembly_-_IBM_Programming_Environment_2.3.pdf, 2005. 2.1
- [87] Intel. 5-Level Paging and 5-Level EPT (White Paper). https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2015. 1, 2.1
- [88] Intel. Remote Action Request White Paper. <https://www.intel.com/content/dam/develop/external/us/en/documents/341431-remote-action-request-white-paper.pdf>, 2021. 2.2.1, 2.3.3
- [89] Intel. Intel trusted domain extensions. <https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf>, 2021. 1, 2.1
- [90] Intel. Intel Virtualization Technology for Directed I/O. <https://edc.intel.com/content/www/xl/es/publications/specification-nuc12dcm-nuc12edb/intel-virtualization-technology-for-directed-i-o/>, 2022. 2.2.1, 2.3.3
- [91] Intel. Intel[®] Optane[™] Persistent Memory. <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html>, 2022. 1, 2.1, 3.1
- [92] Intel. Introducing the Intel Data Streaming Accelerator. <https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator>, 2023. 2.3.3
- [93] Intel. Timed process event based sampling (tpebs), January 2025. URL <https://www.intel.com/content/www/us/en/developer/articles/technical/\time>

d-process-event-based-sampling-tpebs.html. 5.3.5, 5.4.3

- [94] Intel. Intel memory latency checker v3.12, March 2026. URL <https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html>. 4.6.2
- [95] Intel[®]. Itanium[®] Architecture Software Developer’s Manual (Volume 2). <https://www.intel.com/content/www/us/en/products/docs/processors/itanium/itanium-architecture-vol-1-2-3-4-reference-set-manual.html>, 2010. 2.1
- [96] Intel[®]. 64 and IA-32 Architectures Software Developer’s Manual, 2019. 2.3.3, 2.4
- [97] Bruce L. Jacob and Trevor N. Mudge. A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, 1998. 2.1, 3.1, 3.2.2
- [98] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. EXTMEM: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC’24)*, 2024. 3.1
- [99] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems. *ACM Trans. Archit. Code Optim.*, 16(1), mar 2019. ISSN 1544-3566. doi: 10.1145/3309710. URL <https://doi.org/10.1145/3309710>. 3.1
- [100] Joejon Jann, Paul Mackerras, John Ludden, Michael Gschwind, Wade Ouren, Stuart Jacobs, Brian F. Veale, and David Edelsohn. IBM POWER9 system software. *IBM Journal of Research and Development*, 62(4/5), June 2018. 2.1, 3.1, 3.2.2
- [101] jemalloc. jemalloc. <https://jemalloc.net>, 2025. 3.3.1
- [102] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems, HotStorage ’22*, page 45–51, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393997. doi: 10.1145/3538643.3539745. URL <https://doi.org/10.1145/3538643.3539745>. 5.1
- [103] Elena Kakoulli and Herodotos Herodotou. Octopusfs: A distributed file system with tiered storage management. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD ’17*, page 65–78, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3064023. URL <https://doi.org/10.1145/3035918.3064023>. 4.7
- [104] Konstantinos Kanellopoulos, Rahul Bera, Kosta Stojiljkovic, F. Nisa Bostanci, Can Firtina, Rachata Ausavarungnirun, Rakesh Kumar, Nastaran Hajinazar, Mohammad Sadrosadati, Nandita Vijaykumar, and Onur Mutlu. Utopia: Fast and efficient address translation via hybrid restrictive & flexible virtual-to-physical address mappings. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’23*, page 1196–1212, New York, NY, USA, 2023. Associa-

- tion for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3623789. URL <https://doi.org/10.1145/3613424.3623789>. 3.1, 3.2.1
- [105] Konstantinos Kanellopoulos, Hong Chul Nam, Nisa Bostanci, Rahul Bera, Mohammad Sadrosadati, Rakesh Kumar, Davide Basilio Bartolini, and Onur Mutlu. Victima: Drastically increasing address translation reach by leveraging underutilized cache resources. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1178–1195, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614276. URL <https://doi.org/10.1145/3613424.3614276>. 3.1
 - [106] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Energy-efficient address translation. In *Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*, 2016. 2.1, 3.1
 - [107] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*, 2015. 2.1, 2.7, 3.1, 3.8
 - [108] Patrick Kennedy. Hyper-scalers Are Using CXL to Lower the Impact of DDR5 Supply Constraints. <https://www.servethehome.com/hyper-scalers-are-using-cxl-to-lower-the-impact-of-ddr5-supply-constraints-marvell-arm/>, December 2025. 4.1
 - [109] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro*, 43(2): 20–29, March 2023. ISSN 0272-1732. doi: 10.1109/MM.2023.3240774. URL <https://doi.org/10.1109/MM.2023.3240774>. 4.1
 - [110] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro*, 43(2): 20–29, 2023. doi: 10.1109/MM.2023.3240774. 5.1, 5.6
 - [111] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03)*, 2003. 2.1, 3.1
 - [112] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: a single-pass learned index. In *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management*, pages 1–5, 2020. 3.1, 3.2.3, 3.4.2
 - [113] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*, pages 489–504, 2018. 3.1, 3.2.3, 3.4.3, 3.8

- [114] K. R. Krish, Bharti Wadhwa, M. Safdar Iqbal, M. Mustafa Rafique, and Ali R. Butt. On efficient hierarchical storage for big data processing. In *16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, page 403–408, 2016. doi: 10.1109/CCGrid.2016.61. 4.7
- [115] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. Latr: Lazy translation coherence. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18*, page 651–664, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349116. doi: 10.1145/3173162.3173198. URL <https://doi.org/10.1145/3173162.3173198>. 2.2.1, 2.7
- [116] Sandeep Kumar, Aravinda Prasad, and Sreenivas Subramoney. Tierscape: Harnessing multiple compressed tiers to tame server memory tco. In *Proceedings of the 21st European Conference on Computer Systems, EUROSYS '26*, page 247–262, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722127. doi: 10.1145/3767295.3769321. URL <https://doi.org/10.1145/3767295.3769321>. 5.6
- [117] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, 2016. 2.1, 2.6, 2.7, 3.7.3, 3.8
- [118] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *SOSP '17*, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132770. URL <https://doi.org/10.1145/3132747.3132770>. 4.7
- [119] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304053. URL <https://doi.org/10.1145/3297858.3304053>. 4.2.1, 4.4.3, 4.4.4, 4.5.2
- [120] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1.1.1–1.1.8, 2016. doi: 10.1109/IEDM.2016.7838026. 5.1
- [121] 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 Com-

- pating Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613167. URL
- <https://doi.org/10.1145/3600006.3613167>
- . 4.1, 4.7, 5.1, 5.6
- [122] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. A fully associative, tagless dram cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 211–222, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334020. doi: 10.1145/2749469.2750383. URL <https://doi.org/10.1145/2749469.2750383>. 5.3.5
 - [123] 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, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3578835. URL <https://doi.org/10.1145/3575693.3578835>. 1, 4.1, 5.1, 5.2.1, 5.6
 - [124] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: a cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020. 2.4
 - [125] Linux. Control Group v2. <https://docs.kernel.org/admin-guide/cgroup-v2.html>, 2015. 5.4.4
 - [126] Linux. zswap, March 2026. URL <https://docs.kernel.org/admin-guide/mm/zswap.html>. 4.1, 4.7
 - [127] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger, and Huaicheng Li. Dissecting cxl memory performance at scale: Analysis, modeling, and optimization, 2024. URL <https://arxiv.org/abs/2409.14317>. 5.1, 5.6
 - [128] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered memory management beyond hotness. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-47-2. 4.1, 4.7
 - [129] 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. ISBN 9798400723599. doi: 10.1145/3779212.3790201. URL <https://doi.org/10.1145/3779212.3790201>. 4.7
 - [130] Matt Loughran. How Okta Chased Down Severe System CPU Contention in MySQL. <https://developer.okta.com/blog/2015/05/22/tcmalloc>, 2015. 2.1, 3.3.1
 - [131] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. Apex: a high-performance learned index on persistent memory. *Proc. VLDB Endow.*, 15(3): 597–610, November 2021. ISSN 2150-8097. doi: 10.14778/3494124.3494141. URL

<https://doi.org/10.14778/3494124.3494141>. 3.8

- [132] Yongping Luo, Peiquan Jin, Zhaole Chu, Xiaoliang Wang, Yigui Yuan, Zhou Zhang, Yun Luo, Xufei Wu, and Peng Zou. Morphtree: a polymorphic main-memory learned index for dynamic workloads. *The VLDB Journal*, pages 1–20, 2023. 3.1, 3.2.3
- [133] Piotr R. Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC’06)*, 2006. 3.6.2
- [134] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvine, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and misconceptions around reducing carbon embedded in cloud platforms. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems, HotCarbon ’23*, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702426. doi: 10.1145/3604930.3605717. URL <https://doi.org/10.1145/3604930.3605717>. 4.2.2
- [135] Xiaoyu Ma and David Patterson. Challenges and research directions for large language model inference hardware. *Computer*, 59(5):55–64, May 2026. ISSN 1558-0814. doi: 10.1109/mc.2026.3652916. URL <http://dx.doi.org/10.1109/MC.2026.3652916>. 1
- [136] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020. 2.1, 2.6, 2.7, 3.8
- [137] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2021, page 28–38, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384483. doi: 10.1145/3459898.3463905. URL <https://doi.org/10.1145/3459898.3463905>. 2.1, 2.6, 2.7, 3.8
- [138] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafir. rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’15, pages 355–368, New York, NY, USA, March 2015. Association for Computing Machinery. ISBN 9781450328357. doi: 10.1145/2694344.2694355. URL <https://doi.org/10.1145/2694344.2694355>. 2.7
- [139] Tobias Mann. DRAM prices expected to double in Q1 as AI ambitions push memory fabs to their limit. https://www.theregister.com/2026/02/02/dram_prices_expected_to_double/, February 2026. 4.1
- [140] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K.

- John. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*, Cambridge, MA, USA, October 2017. 2.7
- [141] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Virtual address translation via learned page table indexes. In *Workshop on ML for Systems at NeurIPS*, 2018. 3.8
 - [142] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’52, page 1023–1036, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358294. URL <https://doi.org/10.1145/3352460.3358294>. 2.1, 3.1, 3.7, 3.7.5
 - [143] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019. 2.1, 2.6, 2.7
 - [144] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkupati, 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 *SOSP ’19*, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359657. URL <https://doi.org/10.1145/3341301.3359657>. 2.2.5
 - [145] 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. ISBN 9781450399180. doi: 10.1145/3582016.3582063. URL <https://doi.org/10.1145/3582016.3582063>. 1, 4.1, 4.3.2, 4.4.5, 4.6, 4.6.1, 4.6.3, 4.7, 5.1, 3, 5.2.1, 5.3.6, 5.4.1, 5.4.3, 5.4.3, 5.5.1, 5.6
 - [146] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014. 2.2.1
 - [147] Jai Menon, D. A. Pease, Robert M. Rees, L. Duyanovich, and B. Hillsberg. Ibm storage tank - a heterogeneous scalable san file system. *IBM Systems Journal*, 42(2): 250–267, 2003. doi: 10.1147/sj.422.0250. 4.7
 - [148] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jimenez. CHiRP: Control-Flow History Reuse Prediction. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*, 2020. 2.7
 - [149] MongoDB. Transparent Huge Pages Warning. <https://github.com/mongodb/mon>

- go/blob/eeee7c2f80bdaf49a197a1c8149d7bc6cbc9395e/src/mongo/db/startup_warnings_mongod.cpp, 2018. 2.1
- [150] Timothy Prickett Morgan. CXL and Gen-Z iron out a coherent interconnect strategy. <https://www.nextplatform.com/connect/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/1644243>, April 2020. 1, 4.1
 - [151] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. Graph-BIG: Understanding Graph Computing in the Context of Industrial Solutions. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC’15)*, 2015. 3.6.2
 - [152] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010. ISSN 0163-5980. doi: 10.1145/1713254.1713276. URL <https://doi.org/10.1145/1713254.1713276>. 5.1
 - [153] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26, 2009. 4.7
 - [154] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, May 2004. 3.2.2
 - [155] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’18)*, 2018. 2.1, 2.2.4, 2.6, 2.7, 3.8
 - [156] Ashish Panwar, Sorav Bansal, and K. Gopinath. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’19)*, 2019. 2.1, 2.6, 2.7, 3.1, 3.8
 - [157] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA’17)*, 2017. 2.7
 - [158] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Every walk’s a hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’22, page 128–141, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507718. URL <https://doi.org/10.1145/3503222.3507718>. 3.1, 3.7, 3.7.5, 3.8
 - [159] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafir. Utilizing the

- IOMMU scalably. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 549–562, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL <https://www.usenix.org/conference/atc15/technical-session/presentation/peleg>. 2.2.1
- [160] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*, 2012. 2.1, 2.6, 2.7, 3.1, 3.8
 - [161] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA’14)*, 2014. 2.7, 3.8
 - [162] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways? In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, Waikiki, Hawaii, USA, December 2015. 2.1, 2.6, 3.1, 3.8
 - [163] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafir. The benefits of general-purpose on-NIC memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1130–1147, Lausanne, Switzerland, February 2022. 2.2.1
 - [164] PMem.io. Two-level memory, January 2025. URL <https://pmem.io/glossary/#2lm>. 5.3.5
 - [165] Raghavendra Pradyumna Pothukuchi, Amin Ansari, Petros Voulgaris, and Josep Torrellas. Using multiple input, multiple output formal control to maximize resource efficiency in architectures. *ACM SIGARCH Computer Architecture News*, 44(3): 658–670, 2016. 4.7
 - [166] DPDK Project. Memory in dpdk, part 1: General concepts. <https://www.dpdk.org/blog/2019/08/21/memory-in-dpdk-part-1-general-concepts/>, 2019. 2.2.5
 - [167] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, page 31–47, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314637. URL <https://doi.org/10.1145/3314221.3314637>. 3.3.1
 - [168] Stratos Psomadakis, Chloe Alverti, Vasileios Karakostas, Christos Katsakioris, Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Elastic Translations: Fast Virtual Memory with Multiple Translation Sizes. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages

- 17–35, Los Alamitos, CA, USA, November 2024. IEEE Computer Society. doi: 10.1109/MICRO61859.2024.00012. URL <https://doi.ieeeecomputersociety.org/10.1109/MICRO61859.2024.00012>. 3.1, 3.8
- [169] Moinuddin K. Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 775–787, Fukuoka, Japan, October 2018. IEEE Press. ISBN 9781538662403. doi: 10.1109/MICRO.2018.00068. URL <https://doi.org/10.1109/MICRO.2018.00068>. 2.3.3
- [170] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing architectural resources for all page sizes in x86 processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’21, page 1106–1120, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480062. URL <https://doi.org/10.1145/3466752.3480062>. 2.7, 3.8
- [171] 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. ISBN 9781450387095. doi: 10.1145/3477132.3483550. URL <https://doi.org/10.1145/3477132.3483550>. 4.1, 5.6
- [172] Will Reese. Nginx: The high-performance web server and reverse proxy. *Linux J.*, 2008. 2.4
- [173] 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. ISBN 9798400704376. doi: 10.1145/3627703.3650075. URL <https://doi.org/10.1145/3627703.3650075>. 4.1, 5.1, 5.6
- [174] Facebook Research. DCPerf benchmark suite for hyperscale cloud applications. <https://github.com/facebookresearch/DCPerf>, 2025. 3.3.1, 4.1, 5.5.1
- [175] Arun F. Rodrigues, Jeanine Cook, Elliott Cooper-Balis, K. Scott Hemmert, Chad Kersey, Rolf Riesen, Paul Rosenfeld, Ron Oldfield, and Marlow Weston. The Structural Simulation Toolkit. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC’10)*, Tampa, Florida, November 2006. 3.6.1
- [176] Arun F Rodrigues, Gwendolyn Renae Voskuilen, Simon David Hammond, and Karl Scott Hemmert. Structural simulation toolkit (sst). Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016. 2.4
- [177] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010. doi: 10.1109/HPCA.2010.5416643. 2.2.1,

- [178] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, Jan 2011. ISSN 1556-6056. doi: 10.1109/L-CA.2011.4. 3.6.1
- [179] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows internals, part 2*. Pearson Education, 2012. 2.2.1
- [180] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA’17)*, Toronto, ON, Canada, June 2017. 2.1, 2.7, 3.1
- [181] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. Can learned models replace hash functions? *Proceedings of the VLDB Endowment*, 16(3):532–545, 2022. 3.1, 3.2.3
- [182] Samsung. Samsung Electronics Introduces Industry’s First 512GB CXL Memory Module. <https://news.samsung.com/global/samsung-electronics-introduce-s-industrys-first-512gb-cxl-memory-module>, 2019. 1, 2.1, 3.1
- [183] Harald Servat, Antonio J. Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 126–136, 2017. doi: 10.1109/CLUSTER.2017.50. 5.3.5
- [184] 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. ISBN 9781450394871. doi: 10.1145/3552326.3587449. URL <https://doi.org/10.1145/3552326.3587449>. 4.1, 5.1, 5.6
- [185] Rajath Shashidhara, Simon Peter, Scott Hare, and Kimberly Keeton. Closing the benchmark gap for tiered memory. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems, DIMES ’25*, page 45–53, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400722264. doi: 10.1145/3764862.3768177. URL <https://doi.org/10.1145/3764862.3768177>. 4.1, 4.1, 4.6
- [186] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. Babelfish: Fusing address translations for containers. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, ISCA 2020, 2020. doi: 10.1109/ISCA45697.2020.00049. 2.7
- [187] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20*, 2020. 3.1, 3.2.2, 3.3.2, 3.4.4, 3.6.3, 3.8

- [188] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020. 2.1, 2.3.3, 2.6
- [189] S. Skogestad and I. Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. Wiley, Chichester, UK, 2005. ISBN 9780470011676. URL <https://books.google.com/books?id=97iAEAAAQBAJ>. 4.4.4, 4.7
- [190] J. E. Smith and Ravi Nair. The Architecture of Virtual Machines. *IEEE Computer*, 38(5):32–38, 2005. doi: 10.1109/MC.2005.173. 2.1, 3.1
- [191] Shekhar Srikantaiah and Mahmut Kandemir. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*, 2010. 2.7
- [192] Ross Stenfort, Ta-Yu Wu, Lee Prewitt, Paul Kaler, David DeRosa, William Lynn, and Austin Bolen. OCP Datacenter NVMe® SSD Specification v2.0. <https://www.opencompute.org/documents/datacenter-nvme-ssd-specification-v2-0r21-pdf>, July 2021. 4.6
- [193] Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Parallel virtualized memory translation with nested elastic cuckoo page tables. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, 2022. doi: 10.1145/3503222.3507720. 2.1, 2.6, 3.1, 3.2.2, 3.8
- [194] Jovan Stojkovic, Namrata Mantri, Dimitrios Skarlatos, Tianyin Xu, and Josep Torrellas. Memory-efficient hashed page tables. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1221–1235, 2023. doi: 10.1109/HPCA56546.2023.10071061. 2.1, 2.6, 3.1, 3.2.2, 3.8
- [195] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaflet: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 442–456, 2023. 3.8
- [196] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxian Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614256. URL <https://doi.org/10.1145/3613424.3614256>. 4.1, 5.1, 5.6
- [197] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, 1994. 2.1, 3.1, 3.2.2

- [198] Madhusudhan Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, page 171–182, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916603. doi: 10.1145/195473.195531. URL <https://doi.org/10.1145/195473.195531>. 2.1, 3.1
- [199] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Trade-offs in Supporting Two Page Sizes. In *19th International Symposium on Computer Architecture (ISCA’92)*, 1992. 2.7, 3.8
- [200] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. Xindex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 308–320, 2020. 3.1
- [201] Mellanox Technologies. Introducing 200G HDR InfiniBand Solutions. <https://network.nvidia.com/files/doc-2020/wp-introducing-200g-hdr-infiniband-solutions.pdf>, 2019. 2.2.5
- [202] Mellanox Technologies. Mellanox Adapters Programmer’s Reference Manual. <https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf>, 2020. 2.2.1
- [203] The Linux Kernel Archives. Psi - pressure stall information. <https://docs.kernel.org/accounting/psi.html>, 2018. 4.4.3, 4.5.1
- [204] The Linux Kernel Archives. Transparent hugepage support. <https://www.kernel.org/doc/Documentation/vm/transhuge.txt>, 2019. 2.1, 2.2.1
- [205] The Linux Kernel Archives. x86 virtual memory map. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2023. 2.2.1
- [206] The Linux Kernel Archives. Hugetlb pages. <https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html>, 2023. 2.2.3
- [207] Chandrahas Tirumalasetty and Narasimha Reddy Annapareddy. Contention aware dram caching for cxl-enabled pooled memory. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS ’24, page 157–171, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400710919. doi: 10.1145/3695794.3695808. URL <https://doi.org/10.1145/3695794.3695808>. 4.1, 5.6
- [208] TrendForce. DRAM Industry Analysis–1Q26. <https://www.trendforce.com/research/download/RP260305JB>, March 2026. 4.1
- [209] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. Morrigan: A composite instruction tlb prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’21, page 1138–1153, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480049. URL <https://doi.org/10.1145/3466752.3480049>. 3.1, 3.8

- [210] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. Exploiting page table locality for agile tlb prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 85–98, 2021. doi: 10.1109/ISCA52012.2021.00016. 2.1, 2.6, 2.7, 3.1, 3.8
- [211] Carlos Villavieja, Vasileios Karakostas, Lluís Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349, 2011. doi: 10.1109/PACT.2011.65. 2.2.1, 2.7
- [212] 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. ISBN 9798400712517. doi: 10.1145/3694715.3695968. URL <https://doi.org/10.1145/3694715.3695968>. 4.1, 4.4.3, 4.7, 5.1, 5.6
- [213] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvène, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warriar, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing cloud servers for lower carbon. In *Proceedings of the 51st Annual International Symposium on Computer Architecture, ISCA '24*, page 452–470, Buenos Aires, Argentina, 2025. IEEE Press. ISBN 9798350326581. doi: 10.1109/ISCA59077.2024.00041. URL <https://doi.org/10.1109/ISCA59077.2024.00041>. 4.1, 4.2.2
- [214] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. *ACM SIGARCH computer architecture news*, 37(3):314–324, 2009. 4.7
- [215] Ziqi Wang, Kaiyang Zhao, Pei Li, Andrew Jacob, Michael Kozuch, Todd Mowry, and Dimitrios Skarlatos. Memento: Architectural support for ephemeral memory management in serverless environments. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 122–136, 2023. 3.3.1
- [216] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022*, 2022. doi: 10.1145/3503222.3507731. 1, 2.3.2, 4.1, 4.2.1, 4.4.3, 4.6, 4.6.1, 4.7
- [217] Wikipedia contributors. Granite Rapids, 2026. URL https://en.wikipedia.org/wiki/Granite_Rapids. 4.2.2
- [218] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer*

- [219] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. In *Proceedings of the VLDB Endowment*, pages 1276–1288, 2021. doi: 10.14778/3457390.3457393. URL <https://dl.acm.org/doi/10.14778/3457390.3457393>. 3.1, 3.2.3
- [220] Kai Wu, Yingchao Huang, and Dong Li. Unimem: runtime data management on non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17*, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351140. doi: 10.1145/3126908.3126923. URL <https://doi.org/10.1145/3126908.3126923>. 5.3.5
- [221] www.7-cpu.com. Intel Skylake Timing. <https://www.7-cpu.com/cpu/Skylake.html>, 2023. 1, 2.1, 3.1
- [222] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22*, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519556. URL <https://doi.org/10.1145/3492321.3519556>. 5.1
- [223] 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 *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation, OSDI'24*, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3. 4.1, 5.1, 5.6
- [224] Zi Yan. 1GB THP support on x86_64. <https://lwn.net/Articles/830435/>, 2020. 2.7
- [225] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA'19)*, Phoenix, AZ, USA, June 2019. 2.1, 2.2.1, 2.2.4, 2.6, 2.7, 3.8
- [226] 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. ISBN 9781450362405. doi: 10.1145/3297858.3304024. URL <https://doi.org/10.1145/3297858.3304024>. 4.1, 5.1, 5.6
- [227] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies, FAST'20*, page 169–182, USA, 2020. USENIX Association. ISBN 9781939133120. 5.1
- [228] Idan Yaniv and Dan Tsafir. Hash, Don't Cache (the Page Table). In *Proceed-*

- ings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS'16)*, 2016. 2.1, 2.6, 3.1, 3.2.2
- [229] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015. URL <https://eprint.iacr.org/2015/905>. 2.3.3
 - [230] 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, ISCA '25*, page 405–418, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712616. doi: 10.1145/3695053.3731001. URL <https://doi.org/10.1145/3695053.3731001>. 4.1, 5.6
 - [231] Huang Ying. Numa balancing: optimize memory placement for memory tiering system, January 2025. URL <https://lwn.net/Articles/849095/>. 5.1
 - [232] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: bandwidth-efficient dram caching via software/hardware cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-50 '17*, page 1–14, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349529. doi: 10.1145/3123939.3124555. URL <https://doi.org/10.1145/3123939.3124555>. 5.3.5
 - [233] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016. ISSN 0001-0782. doi: 10.1145/2934664. URL <https://doi.org/10.1145/2934664>. 5.5.3
 - [234] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. Direct memory translation for virtualized clouds. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '24*, page 287–304, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640358. URL <https://doi.org/10.1145/3620665.3640358>. 3.1, 3.8
 - [235] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA '23*, pages 1–15, New York, NY, USA, June 2023. ISBN 9798400700958. doi: 10.1145/3579371.3589079. URL <https://dl.acm.org/doi/10.1145/3579371.3589079>. 1
 - [236] Kaiyang Zhao, Yuang Chen, Xenia Xu, Dan Schatzberg, Nastaran Hajinaza, Rupin Vakhawala, Andy Anderson, and Dimitrios Skarlatos. Learning to walk: Architecting learned virtual memory translation. In *Proceedings of the 58th IEEE/ACM*

International Symposium on Microarchitecture, MICRO '25, page 1777–1792, 2025.
1

- [237] Kaiyang Zhao, Neha Gholkar, Hasan Maruf, Abhishek Dhanotia, Johannes Weiner, Gregory Price, Ning Sun, Bhavya Dwivedi, Stuart Clark, and Dimitrios Skarlatos. Equilibria: Fair multi-tenant cxl memory tiering at scale, 2026. URL <https://arxiv.org/abs/2602.08800>. 1
- [238] 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 *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3. 4.1, 4.7, 5.1, 5.3.5, 5.6
- [239] 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 *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3. 4.2.2
- [240] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. Neomem: Hardware/software co-design for cxl-native memory tiering. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, MICRO '24, page 1518–1531, Austin, TX, USA, 2024. IEEE Press. doi: 10.1109/MICRO61859.2024.00111. URL <https://doi.org/10.1109/MICRO61859.2024.00111>. 4.1, 4.7, 5.6
- [241] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423. doi: 10.1145/2785956.2787484. URL <https://doi.org/10.1145/2785956.2787484>. 2.2.5
- [242] K.J. Åström and T. Hägglund. The future of pid control. *Control Engineering Practice*, 9(11):1163–1175, 2001. ISSN 0967-0661. doi: [https://doi.org/10.1016/S0967-0661\(01\)00062-4](https://doi.org/10.1016/S0967-0661(01)00062-4). URL <https://www.sciencedirect.com/science/article/pii/S0967066101000624>. PID Control. 4.4.4, 4.7