Understanding and Optimizing Tiered Memory and Storage Systems

A Dissertation Presented

by

Tyler Estro

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-26-03

August 2026

Stony Brook University
The Graduate School

Tyler Estro

We, the dissertation committee for the above candidate for
the degree of Doctor of Philosophy, hereby recommend
acceptance of this dissertation.

Erez Zadok - Dissertation Advisor
Professor, Computer Science Department

Michael Ferdman - Chairperson of Dissertation Committee
Associate Professor, Computer Science Department

Anshul Gandhi
Associate Professor, Computer Science Department

Carl Waldspurger
Principal, Carl Waldspurger Consulting

This dissertation is accepted by the Graduate School

Celia Marshik
Dean of the Graduate School

Abstract of the Dissertation

Understanding and Optimizing Tiered Memory and Storage Systems

by

Tyler Estro

Doctor of Philosophy

in

Computer Science

Stony Brook University

2026

Tiered memory and storage systems combine heterogeneous devices to balance competing objectives, such as performance, cost, and media endurance. However, these systems include many parameters and design choices that compound with each additional tier, creating a vast configuration space that is difficult to evaluate, or search efficiently for optimal configurations. Furthermore, as technologies continue to evolve, they expand the space and create new tiering opportunities that require new techniques to exploit. We address these challenges on three fronts: comprehensive multi-tier evaluation, efficient exploration of configuration spaces, and a novel tiering mechanism for next-generation systems.

We first examine trends in cache analysis and find that many techniques are limited, focusing primarily on performance, analyzing tiers in isolation, and overlooking key trade-offs. To address these limitations, we develop a multi-tier cache simulator that evaluates configurations across a broad range of parameters and produces diverse metrics. Simulations of real-world traces show that multi-tier analysis can overturn common assumptions about device selection, hierarchy complexity, and policy effectiveness. We next present a framework for efficiently exploring large multi-tier configuration spaces. It combines hash-based sampling, curve simplification, knee detection, and post-processing filters to select key points in cache miss ratio curves, narrowing the search to promising con-

figurations. Relatedly, we also introduce Z-Method, a multi-knee detection technique that uses statistical outlier detection to robustly choose significant points. Our framework accurately identifies optimal configurations while substantially reducing simulation requirements. Finally, we present Pontoon, which implements tiering with Compute Express Link (CXL) shared memory to accelerate live virtual machine migration. Pontoon demotes and remaps guest memory from local DRAM to CXL in a single pass, avoiding dirty-page retransmission. At the destination, execution resumes directly from shared memory while a tiering policy uses migration telemetry to promote hot pages to local DRAM in the background. Pontoon reduces migration time, blackout duration, and data movement, and completes migration under workloads where traditional migration fails to converge.

It is our thesis that understanding and optimizing tiered memory and storage systems requires comprehensive evaluation of multi-tier configurations, efficient exploration of configuration spaces, and novel tiering mechanisms that exploit opportunities introduced by evolving technologies.

To my family

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

As I reach the end of my Ph.D., I find myself with a long list of people to thank for enabling me to persevere and succeed in ways that would not otherwise have been possible. It was a long, bumpy road filled with many unexpected hardships that I stumbled and improvised my way through, and I was able to reach the end of it thanks to them.

First, I sincerely thank my advisor, Prof. Erez Zadok, for his ability to masterfully balance being accommodating with the discerning application of pressure, enabling me to consistently make progress. The stern, authoritative presence he maintains to cultivate professionalism belies his warm-hearted nature and deep passion for helping all of his students achieve their goals. He guided me through one of the greatest challenges of my life in a way I do not think anyone else could have, and I will forever be grateful for his wisdom and understanding. I also thank Martha Zadok for her extremely warm support, which has always been felt and greatly appreciated. I wish both of you and your wonderful family good health, happiness, and prosperity.

To all of my academic and industry collaborators. Prof. Geoff Kuenning, for his easy-going and witty personality and ability to contribute to any topic with his deep command of the fundamentals and uniquely "Geoff" skillset. Dr. Carl Waldspurger, for his vast expertise and consistency in identifying and deeply probing difficult technical problems. Prof. Anshul Gandhi, for being a well-established scholar yet always treating me as a peer. Prof. Mário Antunes, for filling in the gaps in my research with his expertise and always being extremely supportive. Prof. Michael Ferdman, for always bringing his own distinct perspective and battling it out in the trenches with me to meet seemingly impossible deadlines. Prof. Heather Lynch and all of the IACS folks, for providing me with many opportunities, experiences, and encouragement. I'd also like to thank Professors Peter Desnoyers, Klaus Mueller, and Avani Wildani for all of their support and our many meaningful collaborations.

To all of the brilliant students I've had the pleasure of working with. Zhen Cao, who mentored me when I first joined the program. Yifei Liu, whom I worked closely with on the first project that I led and will always consider a good friend. Pranav Bhandari, Sri Pramodh Rachuri, Anjul Tyagi, Muhammad Wajahat, and Xinyu Zhang, for all of the projects that they led and that I was fortunate to have been a part of. To all of my lab mates at FSL and the many graduate and undergraduate students that I worked with over the years.

To my family. I hope that reaching this point has made them proud and, in some way, reflects everything we have overcome together.

My mother Patricia, for always being there to reassure and support me. For always putting her children first through the toughest of times and teaching me by example that when everything falls apart, you can pick up the pieces and still realize your dreams.

My father Chuck, who passed away during my Ph.D., for his excessive pride in me and the hard lessons I learned through him that molded me into the man I am today. For instilling the confidence and ambition in me that are necessary to pursue great heights.

My brother Chuck, for his utter commitment to benevolence. For his never-ending struggle to find the right expression, yet being the most dependable, noble, and consistent person I know.

My sister Tricia, who also passed away during my Ph.D., for providing perspective and strengthening my resolve. You will always be missed.

To my friends who stuck with me through all these years: Chris Troise, Adam Smith, Brian Smith, Anand Aiyer, Jayesh Ranjan, and the many others who have supported me and shared experiences with me along the way.

I also gratefully acknowledge the institutions and funding agencies that supported my research throughout my doctoral studies.

This work was made possible in part through support from Microsoft Azure, Dell-EMC, NetApp, IBM, and Facebook, as well as a SUNY/IBM Alliance award and a Stony Brook OVPR Seed Grant. Computational resources used in related work were provided through the SeaWulf computing system by Stony Brook Research Computing and Cyberinfrastructure and the Institute for Advanced Computational Science at Stony Brook University. SeaWulf was made possible through National Science Foundation (NSF) award OAC-1531492. Related educational work was supported by NSF awards OAC-2230077, OAC-2230078, and OAC-2230079 and used the Chameleon testbed, which is supported by NSF award CNS-2431425.

Research presented in this dissertation and related work conducted during my

doctoral studies were supported by NSF under awards
IIS-1527200, CCF-1918225, IIS-1941613, CNS-1251137, CNS-1302246, CNS-1305360, CNS-1622832, CNS-1650499, CNS-1729939, CNS-1730726, CNS-1750109, CNS-1755958, CNS-1900589, CNS-1900706, CNS-1910327, CNS-1951880, CNS-2106263, CNS-2106434, and CNS-2214980, as well as NSF SBIR contract 1926949. Additional support was provided by the Office of Naval Research under award N00014-16-1-2264.

Related work was also supported by FCT – Fundação para a Ciência e a Tecnologia, I.P., through UIDB/50008/2020 (DOI 10.54499/UIDB/50008/2020) and UIDP/50008/2020 (DOI 10.54499/UIDP/50008/2020), with national funds and, where applicable, European Union co-funding.

Chapter 1

Introduction

Modern memory and storage hierarchies have become increasingly complex, forming multi-tier systems comprising heterogeneous devices in a wide range of topologies. From operating system page caches to large-scale distributed systems, tiering is employed to balance competing objectives, including performance, cost, and media endurance. These systems dynamically reallocate cache capacity or move data among tiers as workload demand changes, improving utilization and helping meet performance objectives [174, 82, 58, 31, 32]. The configuration space for tiering systems is vast, spanning device types, numbers of tiers, tier capacities, tiering policies, and their tunable parameters. These configurations are difficult to evaluate comprehensively or search efficiently, particularly as workload locality and reuse behavior vary over time [131, 157, 125, 11, 28, 161]. Within this evolving landscape, Compute Express Link (CXL) introduces byte-addressable memory expansion, pooling, and sharing across hosts [34]. CXL further expands the configuration space and creates tiering opportunities that require new techniques to exploit.

This dissertation addresses three challenges of tiering. The first is the comprehensive evaluation of multi-tier configurations and their trade-offs. The second is the efficient exploration of large configuration spaces to identify high-quality configurations. The third is the effective exploitation of tiering opportunities introduced by evolving technologies.

First, to address the challenge of multi-tier evaluation, we examined cache analysis trends and identified key limitations in existing evaluation techniques [44]. We found that prior techniques often focused solely on performance, analyzed tiers in isolation, and ignored trade-offs such as cost versus performance. Existing cache simulators were similarly restrictive, supporting only fixed hier-

archies and limited metrics. We addressed these gaps by extending PyMimircache [165] to create a general n-level cache simulator capable of modeling arbitrary hierarchies, capturing both performance and cost, and enabling the analysis of trade-offs across multiple metrics. We uncovered surprising insights through simulations using real-world traces: (a) when total cost was held constant, lower-priced DRAM outperformed high-end DRAM by providing greater capacity; (b) aging SSDs reduced performance enough to favor simpler hierarchies; and (c) write-back policies delivered up to 6× the throughput of write-through on identical hardware.

Next, to address the challenge of exploring large multi-tier configuration spaces, we developed a framework that narrows the search to promising configurations [42]. Miss ratio curves (MRCs) are common analysis tools for evaluating cache performance, but generating them at fine granularity for every tier and configuration can be prohibitively expensive. Our approach focuses on identifying knee points in MRCs, where a small increase in cache size yields a disproportionately large drop in miss ratio, indicating promising candidates for further evaluation. The framework applies hash-based sampling [150] and curve simplification [121], and recursively applies single-knee detection methods to identify multiple knee points. In addition, we introduced a novel multi-knee detection algorithm called Z-Method that employs statistical outlier detection to identify knee points robustly and efficiently. We evaluated our framework using 106 diverse real-world workloads and reduced the number of simulations needed to find optimal two-tier hierarchies by 5.5× for ARC and 7.7× for LRU. We also applied our framework to seed the initial populations of evolutionary algorithms used to optimize multi-tier cache configurations, accelerating convergence by 34% across a broad range of configurations and datasets.

Lastly, to address the challenge of exploiting tiering opportunities introduced by evolving technologies, we developed Pontoon, a rack-local fast path for live virtual machine (VM) migration that uses shared CXL memory. Traditional pre-copy migration iteratively retransmits guest pages dirtied after they are copied, increasing migration time and data movement and potentially preventing migration from converging [114, 33]. Pontoon is implemented in QEMU/KVM and provides transparent guest memory tiering while performing migration in three phases: single-pass demotion, stop-and-flush, and background promotion. During demotion, the source copies each DRAM-resident page to shared CXL memory at most once and remaps it so that subsequent guest writes update the shared memory directly, eliminating dirty-page retransmission. Pontoon operates on VMs in any tiering state, so pages already resident in CXL require no data movement

during migration. Once demotion is complete, Pontoon pauses the VM, flushes cached guest data to CXL, transfers CPU and device state, and resumes the VM directly from shared CXL memory on the destination. After resumption, the destination applies a tiering policy that retains pages in CXL or promotes selected pages to local DRAM in the background, using migration telemetry to prioritize recently modified pages. For 64 GB VMs, Pontoon improves migration time by 2.1–9.4$\times$ over QEMU's TCP- and RDMA-based pre-copy paths, reduces blackout from 267–281 ms to 29 ms, and completes migration at dirty-page rates where traditional network-based pre-copy fails to converge.

It is our thesis that understanding and optimizing tiered memory and storage systems requires comprehensive evaluation of multi-tier configurations, efficient exploration of configuration spaces, and novel tiering mechanisms that exploit opportunities introduced by evolving technologies. This dissertation supports this thesis through three contributions: a multi-tier cache simulator for comprehensive evaluation, a key point selection framework for efficient exploration of configuration spaces, and Pontoon, a CXL-based tiering mechanism for live VM migration.

The rest of this dissertation is organized as follows: Chapter 2 provides background and motivation for the challenges addressed. Chapter 3 reviews prior work related to the systems and techniques presented here. Chapter 4 examines cache analysis techniques and presents our multi-tier cache simulator. Chapter 5 presents our framework for efficiently exploring multi-tier cache configurations using knee detection. Chapter 6 presents the design, implementation, and evaluation of Pontoon. Chapter 7 concludes by summarizing the contributions and discussing directions for future research.

Chapter 2

Background and Motivation

This chapter presents the background and motivation for the three challenges addressed in this dissertation. We first examine why multi-tier configurations require comprehensive evaluation, then describe the difficulty of exploring their large configuration spaces efficiently, and finally discuss the challenge of exploiting new tiering opportunities introduced by evolving technologies, focusing on Compute Express Link (CXL) and the use of shared CXL memory for live virtual machine (VM) migration.

2.1 Comprehensive Multi-Tier Evaluation

Tiered memory and storage systems can comprise many devices and configurable components, making them difficult to evaluate comprehensively. Configurations can vary in the number and arrangement of tiers, device types and models, tier capacities, caching policies, and their tunable parameters. New memory and storage technologies introduce devices with different cost, capacity, and performance characteristics. Workload characteristics such as data reuse, access patterns, and working-set behavior affect cache demand and can change over time [157, 11]. Virtualized and cloud environments also make it increasingly practical to reallocate cache resources dynamically as workload demands change [58, 32]. Together, these factors create a large and evolving configuration space that popular cache analysis techniques do not fully evaluate.

Miss ratio curve (MRC) analysis is widely used to characterize how cache performance changes with capacity. For a given workload and replacement algorithm, an MRC plots the miss ratio as a function of cache size. Systems can use

MRCs to guide cache partitioning and adjust allocations dynamically as workload conditions change [62, 19]. Although useful, an MRC typically characterizes the performance of only one tier of cache under a particular reference stream. This limits its ability to characterize the interactions and trade-offs that emerge across multi-tier configurations.

Even when multiple tiers are considered, evaluation remains incomplete if it is limited to a narrow set of conventional performance metrics. Hit ratio, average latency, and throughput provide useful summaries, but they can obscure important behavior within the hierarchy. Tail latency helps determine whether a configuration can meet service-level objectives [15], inter-tier traffic reveals data-movement overheads and bottlenecks between tiers [115], and write amplification captures hidden write overheads that can reduce performance and device endurance [95]. Cost is also a primary design constraint and should be evaluated using total cost of ownership (TCO), including initial purchase price, energy usage, maintenance, device replacement, and other operational costs incurred over the system's lifetime [89, 90]. Furthermore, compound metrics such as throughput per dollar can reveal trade-offs that are not apparent when performance and cost are considered separately [44].

Evaluating this breadth of configurations and metrics through physical experimentation is costly and time-consuming. Trace-driven simulation provides a practical alternative, but its usefulness depends on the capabilities of the simulator. At the time this work was conducted, publicly available storage-cache simulators generally targeted a single cache or a narrow use case, fixed the number of tiers, omitted important policies, or provided limited analyses [158, 59, 1, 165]. Chapter 4 addresses these limitations by extending PyMimircache into an n-level cache simulator that models arbitrary hierarchies and reports both per-tier and aggregate metrics across a broad range of configurations [44].

2.2 Efficient Exploration of Configuration Spaces

A simulator can reduce the cost of evaluating an individual configuration, but the potential number of configurations in a multi-tier system still makes exhaustive exploration impractical. Even when considering cache capacity alone, the number of possible combinations grows exponentially with each additional tier. These combinations must be evaluated separately because the reference stream observed by each lower tier depends on the cache sizes selected for the tiers above it. Depending on the caching and write policies, the references reaching level $n+1$ may

include misses, write evictions, and flushes from level n. Changing an upper-tier cache size therefore requires a separate simulation to generate the corresponding lower-tier reference stream. For example, a naïve exploration of 100 candidate cache sizes produces 100 configurations for one tier, 100^2 for two tiers, and 100^3 for three tiers, even before varying other configuration parameters. Fine-grained exploration can therefore remain prohibitively expensive even when individual simulations are efficient.

MRCs can be used to explore cache performance across a range of sizes, but doing so efficiently requires both generating them quickly and limiting the number of cache sizes evaluated at each tier. Many techniques reduce MRC construction costs through approximation, sampling, or compact data structures [135, 159, 150, 67, 151]. However, reducing the cost of generating each MRC does not determine which cache sizes should be selected for further exploration. Knees are promising candidates because they occur where a relatively small increase in cache size produces a large reduction in miss ratio. A small number of points from gradually sloping regions may also be selected to ensure adequate exploration. We collectively refer to both types of selected points as *key points*.

Selecting key points is not straightforward because existing knee-detection algorithms do not reliably identify the relevant points in MRCs. These algorithms define knees using different curve properties, including local curvature [84], distance from a reference line [126], piecewise linear fitting [124], and derivative thresholds [9]. Consequently, their results depend strongly on the shape of the curve being analyzed. MRCs may contain multiple significant knees, long gradually sloping regions, and flat or concave regions from which points should be selected carefully or excluded. Their shapes also depend on the replacement algorithm: stack-based algorithms such as LRU satisfy the inclusion property and produce monotonically non-increasing MRCs [102], whereas non-stack algorithms such as ARC [103] can produce non-monotonic curves [151]. Applying existing knee-detection algorithms directly to these varied curves can therefore miss relevant knees or select too many irrelevant points. Reliable key point selection requires methods that identify multiple knees and additional processing that filters and ranks the resulting candidates.

Chapter 5 develops a key point selection framework that combines hash-based sampling, Ramer-Douglas-Peucker curve simplification, multi-knee detection, and post-processing filters to select key points. The framework introduces Z-Method, a novel knee-detection algorithm that uses statistical outlier detection to identify multiple significant knees. Together, these techniques reduce the number of simulations required to explore multi-tier configuration spaces, while accu-

rately identifying optimal two-tier cache configurations in our evaluation [42].

2.3 Exploiting Tiering Opportunities in Evolving Technologies

Evolving technologies can introduce new tiering opportunities and enable mechanisms that were not previously possible. This section describes Compute Express Link (CXL) and how it enables new approaches to live VM migration through memory tiering.

2.3.1 Compute Express Link (CXL)

CXL is an open, industry-supported interconnect standard designed to provide high-performance, cache-coherent memory access among processors, memory expansion devices, and accelerators such as GPUs and FPGAs. Built on the PCIe interface, CXL defines three protocols and classifies devices according to the protocols and memory configurations they support [34]. The three protocols are:

- CXL.io provides non-coherent load/store I/O access and supports operations such as device initialization, discovery, and interrupts.
- CXL.cache allows CXL devices to coherently access and cache host memory.
- CXL.mem allows processors to coherently access memory attached to CXL devices.

CXL devices are classified into three types:

- Type 1 devices are accelerators that use CXL.cache but do not expose device-attached memory through CXL.mem, such as SmartNICs.
- Type 2 devices are accelerators that use both CXL.cache and CXL.mem and expose device-attached memory, such as GPUs and FPGAs.
- Type 3 devices are memory expansion devices that use CXL.mem to expose additional volatile or persistent memory.

CXL creates new opportunities for memory tiering by allowing processors to access CXL-attached memory using ordinary load and store instructions. It supports memory expansion and pooling, allowing memory capacity to be provisioned more flexibly and improving utilization in data centers [86]. Measured CXL-memory access latency is often roughly 2–3$\times$ local-DRAM latency, although the ratio varies substantially with platform, topology, and workload [92, 16]. This performance difference motivates using DRAM and CXL memory as separate tiers. Frequently accessed or latency-sensitive pages can remain in DRAM, while colder pages can be placed in CXL memory to provide additional capacity at a lower performance cost [100, 174, 160].

Shared CXL memory allows multiple hosts to access the same byte-addressable memory region. However, the latest hardware available at the time of this work implements CXL 2.0, which does not support coherent memory sharing across hosts; that capability is defined in CXL 3.0. Software must therefore manage coherence explicitly, ensuring that dirty cache lines are written back and stale cached copies are invalidated before another host accesses the data. Memory sharing further broadens the capabilities of tiered systems and creates opportunities that require new mechanisms to exploit effectively.

2.3.2 Live Virtual Machine Migration

One opportunity enabled by shared CXL memory arises in live VM migration, which transfers a running VM between two physical hosts. Cloud providers use live migration for hardware maintenance, software and firmware updates, load balancing, and resiliency without prolonged service interruption [114, 33, 122, 106]. It is therefore an important data center management mechanism for relocating workloads while maintaining service availability.

Much of a VM's state consists of its memory pages, creating two central challenges. First, as VM memory sizes grow faster than network bandwidth, the increasing amount of guest memory that must be transferred to the destination becomes a central scaling bottleneck. Second, because the VM continues running during much of the migration, its memory contents continue to change while they are being transferred.

Traditional live migration techniques address these challenges by controlling when and how guest memory is transferred over the network. The three primary approaches are pre-copy, post-copy, and hybrid migration. Pre-copy copies guest memory while the VM continues running and iteratively retransmits pages modified during the transfer [114, 33]. Once the remaining dirty set is sufficiently

small, the VM is paused to transfer the remaining pages along with CPU and device state. If pages are dirtied faster than the migration channel can transfer them, the dirty set does not shrink, so migration may fail to converge unless the guest is throttled to force progress [101]. Post-copy instead pauses the VM, transfers CPU and device state, resumes execution on the destination, and fetches guest pages on demand [64]. This approach transfers each page at most once, but places remote page faults directly on the guest's execution path. It also complicates recovery if migration is interrupted [47]. Hybrid migration combines the two approaches by beginning with pre-copy and switching to post-copy if pre-copy does not complete within a configured period [30].

These migration techniques impose two forms of service degradation: *brownout* and *blackout* [72]. During brownout, the migration process contends with the guest VM for CPU, memory, and network resources, reducing guest performance. During blackout, the VM is completely paused before execution resumes at the destination. Different migration techniques make trade-offs among brownout, blackout, retransmission, convergence, and performance after resumption at the destination, but none avoids degradation entirely. Pre-copy is QEMU/KVM's default migration method and the predominant approach used in production by major cloud providers [118, 122, 106].

Shared CXL memory enables a different approach because the source and destination can directly access the same memory tier. Guest pages already placed in CXL need not be transferred during migration. Pages in DRAM can instead be copied to CXL once and remapped so that subsequent guest writes update the shared location directly. The destination can then resume execution from shared CXL memory and promote selected pages to DRAM in the background. Chapter 6 presents Pontoon, which implements tiering over shared CXL memory to accelerate live VM migration. Pontoon uses single-pass demotion and remapping, briefly stops the VM to flush cached guest data before execution resumes at the destination, and promotes hot pages to DRAM in the background using migration telemetry. This design eliminates dirty-page retransmission without placing demand paging on the guest's execution path.

Chapter 3

Related Work

This chapter reviews prior work across five areas relevant to the systems and techniques presented in this dissertation: multi-tier cache evaluation, miss ratio curves, knee detection, Compute Express Link (CXL) tiered memory, and live virtual machine (VM) migration.

3.1 Multi-Tier Cache Evaluation

Cache simulators differ substantially in the systems and levels of abstraction they model. Dinero IV and GCSim use trace-driven simulation to model processor caches [39, 152]. FM-SIM is a functional multicore cache simulator [99]. gem5 simulates complete computer systems with configurable processor, cache, coherence, interconnect, and memory models [18]. These tools are valuable for studying processor and memory behavior, but operate at a different level of abstraction than storage cache analysis.

Simulators designed for storage systems more closely match the challenge addressed in this dissertation. AccuSim models the operating system buffer cache together with kernel prefetching and related I/O behavior [1]. PyMimircache provides trace analysis, cache simulation, and MRC construction for a range of replacement algorithms [165]. Sim-Ideal modeled multiple cache and storage tiers, but fixed the hierarchy to four levels and sent evictions only to the immediately lower tier, preventing it from representing inclusive caching [59]. Pantheon uses a component framework to model I/O systems and storage devices rather than providing a dedicated framework for broad comparisons of cache policies and multi-tier metrics [158]. At the time of the work presented in Chapter 4, these simulators

did not support the combination of arbitrary multi-tier hierarchies, flexible inclusion policies, and diverse evaluation metrics required for comprehensive multi-tier cache analysis.

Later work has developed environments for systems simulation and optimization. DITIS models and optimizes complete file request paths in distributed tiered storage systems [96]. In contrast, the simulator in Chapter 4 focuses on cache behavior across hierarchies with an arbitrary number of levels.

Other work studies multi-tier caching through a particular management policy or optimization objective. MultiCache dynamically partitions heterogeneous cache devices among colocated VMs, whereas REAL coordinates multiple exclusive cache levels using an adaptive, reuse-distance-based replacement algorithm (ReDARC) [120, 28]. CHOPT formulates offline optimal data placement across heterogeneous cache, memory, and storage tiers while accounting for asymmetric access costs [171]. GreenDM evaluates tradeoffs among performance, energy, and endurance in a hybrid drive, and Li *et al.* evaluate the monetary cost of hybrid storage configurations over their expected lifetimes [89, 90]. These systems demonstrate the importance of interactions among tiers and metrics beyond average performance, but each evaluates or optimizes a particular policy, system model, or objective. Chapter 4 instead uses simulation to examine a broad configuration space and shows how conclusions can change when multiple tiers and multiple metrics are evaluated together.

3.2 Miss Ratio Curves (MRCs)

Mattson *et al.* introduced stack distance analysis for constructing exact MRCs for replacement algorithms that satisfy the inclusion property [102]. Subsequent work reduces the time or memory required to construct an MRC for a single cache and reference stream. RapidMRC and FractalMRC estimate MRCs online [135, 62]. CounterStacks uses a compact streaming data structure to approximate MRCs in sublinear space [159]. SHARDS uses uniform randomized spatial sampling to reduce processing and storage overhead [150]. Hu *et al.* accelerate MRC construction for storage caches using the average eviction time (AET) model [67]. Miniature Simulations constructs MRCs for non-stack policies by running scaled simulations at multiple cache sizes [151]. Kosmo generates MRCs simultaneously for several policies that do not satisfy inclusion and uses less memory than Miniature Simulations [128]. These methods are complementary to Chapter 5: they reduce the cost of generating an individual MRC, but do not determine which up-

per tier configurations should be evaluated when each produces a different lower tier reference stream.

MRCs have also been used to guide cache sizing and resource allocation. Cliffhanger estimates local hit rate gradients and detects performance cliffs to guide memory allocation for web caches [32]. mPart uses online MRCs to partition a multitenant key-value store cache [19]. RobinHood reallocates cache capacity to reduce tail latency [15]. QCache combines MRC construction based on locality with cache allocation [49]. These systems use curves to solve particular allocation problems, but do not generate the dependent collection of curves required to explore a general hierarchy of multiple cache tiers.

eMRC is the only prior work that directly approximates multidimensional miss ratio functions for multi-tier caching. It extends Talus's cliff removal approach [14] and relies on cache partitioning, request fractioning, and convex hull approximation to construct a cliff-free miss ratio surface [94]. These requirements tie eMRC to a particular cache model and curve shape. The framework presented in Chapter 5 is more general, leaving replacement policies unchanged and imposing no convexity requirement.

3.3 Knee Detection Algorithms

Knee detection algorithms use different definitions and heuristics to identify points where the behavior of a curve changes sharply. Menger curvature estimates local curvature from three adjacent points [84]. The L-method fits two lines to a curve and selects the split that minimizes their combined error [124]. DFDT derives a threshold from the first derivative to locate a sharp change in slope [9]. The AL and S methods modify the fitting and selection procedures of the L-method [10, 8]. Our evaluation in Chapter 5 found that these differing definitions respond inconsistently to MRC shapes: local curvature can be sensitive to noise, while fitting and threshold methods may miss relevant knees or select weak candidates [42].

Identifying multiple significant knees is necessary for selecting key points from MRCs. The methods above return one selected knee per invocation, whereas Kneedle was designed to detect both single and multiple knees and exposes a sensitivity parameter that affects candidate selection [126]. In Chapter 5's MRC evaluation, Kneedle was sensitive to this parameter and often returned weak or irrelevant candidates, while methods that select a single knee could miss additional significant knees later in a curve [42]. Methods that detect multiple knees also

appear in multi-objective optimization, where knee points identify preferred regions of a Pareto frontier [169]. That setting differs from selecting a sparse set of configurations from an MRC, which may contain multiple knees, gradual slopes, flat regions, noise, and, for non-stack policies, nonmonotonic segments [151].

Chapter 5 addresses this gap by recursively applying algorithms that select one knee to identify multiple knees. It also uses Ramer-Douglas-Peucker curve simplification [121] and introduces sampling, filters applied after detection, selection from gradual regions, and Z-Method, which uses statistical outlier detection to identify multiple significant knees. The resulting framework reduces each MRC to a small set of cache sizes for further exploration.

3.4 CXL Tiered Memory

Studies of real systems characterize the performance differences that motivate CXL memory tiering. Melody systematically characterizes CXL memory across many workloads, devices, latency configurations, and processor platforms, analyzing workload sensitivity, tail latency, and processor tolerance to CXL access latency [92]. Berger *et al.* study the first generation of cloud deployments of CXL memory expansion and identify practical considerations including device validation, monitoring, failure modes, security, and interference among tenants [16]. Weisgut *et al.* evaluate CXL memory for database operations and study how data placement and interleaving across devices affect performance [156]. Unal *et al.* argue that tiered memory systems should combine page placement with techniques that tolerate remaining latency, such as prefetching [143]. These studies characterize or mitigate the cost of CXL accesses, but do not provide a general policy for page placement.

Research on CXL memory expansion has explored different ways of providing and allocating additional memory capacity. DirectCXL prototypes disaggregated memory accessible through loads and stores over CXL.mem [54]. Pond assigns pooled CXL memory to cloud VMs while enforcing limits on workload performance degradation [86].

Page placement and migration between DRAM and CXL memory tiers have been explored through both policy and mechanism design. Transparent Page Placement (TPP) extends Linux NUMA balancing with proactive demotion of cold pages and prompt promotion of hot pages, providing a transparent operating system policy for page placement [100]. Nomad instead focuses on the mechanism used to move pages: it retains shadow copies of promoted pages and uses

transactional page migration to make movement asynchronous and reduce thrashing under pressure on the fast tier [160]. Unlike TPP, Nomad does not determine page temperature itself and relies on existing operating system access tracking to decide which pages should move.

Recent systems improve tiering decisions by incorporating additional performance signals. Colloid shows that the loaded access latency of each tier, rather than only a page's hotness or the unloaded latency specified by the hardware, should guide data placement [149]. Alto uses amortized off-core latency, which incorporates both access latency and memory level parallelism, to suppress migrations whose latency is hidden by overlapping accesses [93]. HybridTier combines access frequency over long periods with recent access momentum and uses compact probabilistic metadata to adapt to changing page activity [129].

Hardware support provides another approach to CXL memory tiering. Intel Flat Memory Mode manages DRAM and CXL placement at the granularity of cache lines inside the memory controller, while Memstrata adds software allocation and performance isolation for tenants that are sensitive to the hardware policy [174]. Collectively, these systems demonstrate broad interest in CXL memory expansion and tiering across hardware and software. Chapter 6 applies CXL memory tiering to live VM migration by using the existing placement of VM pages to reduce data movement.

3.5 Live Virtual Machine Migration

Pre-copy migration iteratively transfers memory while the VM continues running and briefly pauses it for a final stop-and-copy phase [114, 33]. Google's documented production migration system uses pre-copy [122]. Later work improves particular stages of pre-copy. Ibrahim *et al.* control migration using the application's memory rate of change to improve convergence and reduce application impact for HPC workloads with intensive memory use [71]. Haris *et al.* use a learned predictor to stop pre-copy when the predicted downtime satisfies a target [61]. Eswaran *et al.* preserve copy-on-write sharing among related VMs to avoid transferring duplicate pages [45]. Song *et al.* use data and pipeline parallelism across multiple CPU cores and NICs to accelerate migration [130]. These techniques reduce unnecessary transfer or implementation overhead for particular workloads, but retain tracking of dirty pages and may retransmit pages modified while migration is in progress.

Post-copy migration transfers CPU and device state first, resumes the VM on

the destination, and retrieves missing pages on demand [64]. It transfers each page at most once and avoids pre-copy's convergence problem, but remote page faults stall the resumed guest and make recovery more difficult if either host fails before all pages reach the destination. PostCopyFT adds reverse incremental checkpointing and replication to recover a VM after a post-copy failure, at the cost of additional state management and transfer [47]. M3U reduces overheads from kernel memory management, dirty page registration, and fault handling for post-copy migration of large VMs [162]. It improves post-copy scalability, but the destination still fetches missing memory after execution resumes. Hybrid migration begins with pre-copy and later switches to post-copy, bounding pre-copy's convergence delay while retaining demand paging after execution resumes at the destination [72].

RDMA accelerates migration by moving data through an RDMA-capable network interface card (RNIC), reducing CPU and network stack overhead. Huang *et al.* implement live VM migration with RDMA in Xen, using one-sided transfers over InfiniBand to reduce CPU overhead, migration time, and application downtime [68]. Isci *et al.* evaluate fast migration with RDMA using enterprise workloads [74]. Huang *et al.*'s Nomad network virtualization system virtualizes network handles tied to a location and coordinates the transfer of reliable InfiniBand connections that bypass the operating system [69]. Guay *et al.* prototype migration of SR-IOV InfiniBand virtual functions and coordinate network and device reconfiguration during migration [56]. These systems either accelerate the transport or preserve direct device access, but do not change the underlying memory transfer algorithm: pre-copy still retransmits dirty pages, and post-copy still incurs remote faults.

Approaches based on shared memory reduce migration data movement by making VM memory accessible from both hosts. Grapentin *et al.* implement zero-copy VM migration using ThymesisFlow, a memory disaggregation system for IBM POWER9 [55]. This work demonstrates the benefit of leaving memory in place, but depends on a specialized POWER9 and OpenCAPI platform for disaggregated memory. Jo *et al.* implement a QEMU/KVM migration algorithm over software distributed shared memory that avoids copying the VM's entire memory and reduces migration time and transferred data [77]. Their approach provides remote memory and sharing through a DSM layer over the network rather than directly shared CXL memory. The nil-migration proposal describes handing execution between hosts while VM memory remains in shared CXL memory, but the project states that it is at an early stage and does not provide a complete implementation [132]. These systems establish that shared memory can reduce or avoid

copying, but do not provide a QEMU/KVM migration mechanism for available CXL 2.0 hardware. Chapter 6 presents Pontoon, which copies each page resident in DRAM to shared CXL memory at most once, remaps it so subsequent writes reach the shared location, and promotes selected pages to destination DRAM in the background after resumption.

Chapter 4

Desperately Seeking ... Optimal Multi-Tier Cache Configurations

Modern cache hierarchies are tangled webs of complexity. Multiple tiers of heterogeneous physical and virtual devices, with many configurable parameters, all contend to optimally serve swarms of requests between local and remote applications. The challenge of effectively designing these systems is exacerbated by continuous advances in hardware, firmware, innovation in cache eviction algorithms, and evolving workloads and access patterns. This rapidly expanding configuration space has made it costly and time-consuming to physically experiment with numerous cache configurations for even a single stable workload. Current cache evaluation techniques (*e.g.*, Miss Ratio Curves) are short-sighted: they analyze only a single tier of cache, focus primarily on performance, and fail to examine the critical relationships between metrics like throughput and monetary cost. Publicly available I/O cache simulators are also lacking: they can only simulate a fixed or limited number of cache tiers, are missing key features, or offer limited analyses.

It is our position that best practices in cache analysis should include the evaluation of multi-tier configurations, coupled with more comprehensive metrics that reveal critical design trade-offs, especially monetary costs. We developed an n-level I/O cache simulator that models multi-tier cache hierarchies with an arbitrary number of levels, captures detailed statistics for each tier, and supports analysis of performance and cost across configurations. The initial implementation evaluated in this chapter extended an existing cache simulator (PyMimircache). We present several interesting and counter-intuitive results in this chapter.

4.1 Introduction

The vast configuration space of multi-tier caching enables the design of very complex systems. Several tiers of cache and persistent storage can be allocated in numerous arrangements. Moreover, devices can be partitioned into many differently sized cache segments for separate applications. All of these devices can be implemented within, and interact with, any number of independent, large-scale infrastructures (*e.g.*, cloud services, virtual machines, big data warehouses, distributed systems). Furthermore, new storage technologies are constantly emerging (*e.g.*, NVM, 3D flash), introducing additional complexity, greater capacities, and different cost/performance profiles. Our ability to dynamically change hardware in live systems (*e.g.*, adding or deleting RAM, SSD, NVM) has also been increasing, particularly in cloud environments and virtual machines [58, 31, 32], making it significantly easier to reconfigure a cache hierarchy. Workloads continue to evolve as well, with complex and diverse access patterns that affect the frequency of data reuse and the size of working sets, two of the most influential factors in any caching system [131, 157, 125, 11, 28].

Research in cache algorithms and policies is also trying to keep up with these changes. Machine learning and similar techniques that leverage historical data are being incorporated into caching systems to bolster prefetching [166], dynamically switch between replacement algorithms [123, 148, 125], or enhance existing eviction policies [4]. I/O classification has been used to enforce caching policies and improve file system performance [105]. Multi-tier cache eviction algorithms that are aware of some or all layers in the hierarchy at any given time are being developed [28]. The challenges of cache resource allocation and provisioning are being investigated as well [81, 15]. Zhang *et al.* introduced CHOPT, a choice-aware, optimal, offline algorithm for data placement in multi-tier systems [171]. Algorithms such as CHOPT are promising solutions for efficiently finding optimal multi-tier configurations, but their bounding assumptions and inability to model all parameters limit the configuration space they can explore.

Physically experimenting with various cache configurations is costly and time-consuming, with so many parameters to consider (*e.g.*, number of tiers, device types and models, caching policies). A well-known technique for evaluating cache performance without running experiments is Miss Ratio Curve (MRC) analysis [150, 62, 19, 67]. MRCs plot the cumulative miss ratio of all requests in a given workload for some cache eviction algorithm(s) as a function of cache size. Cache size usually ranges from one data block to the size required to store every unique block accessed in the workload, also known as the *working set*. This technique

has many uses, such as comparing eviction algorithms' performance for a given workload or identifying optimal cache size allocations. However, MRCs evaluate the performance of only a single cache and are not capable of accurately modeling the complicated interactions between devices in a multi-tier cache. Recent studies have shown that traditional MRCs are even sub-optimal for resource allocation in a single layer, since they admit data with poor locality into the cache [49].

It is vital that our methods of evaluating caches mature as storage technologies and cache hierarchies continuously evolve. For example, examining performance metrics such as latency or using an MRC to analyze miss ratio as cache size increases may be misleading without also considering the monetary cost of purchasing and using the cache. Cost has a non-linear, positive correlation with cache size, and is fundamentally the primary constraint when deciding how much cache to include in a system. If this were not the case, everyone would cache all data in copious amounts of the fastest DRAM money can buy and back it up with a huge battery. Furthermore, improved performance does not directly translate into cost efficiency, especially in a multi-tier system where devices' cost and performance characteristics can vary wildly. The purchase cost of hardware is a simple example. Ideally, we should be evaluating more comprehensive metrics such as the total cost of ownership, which combines other metrics such as power consumption, the cost of labor to maintain a system, and the projected lifetime of devices given access patterns. It is also essential that we can freely evaluate the relationship between metrics (*e.g.*, throughput/$) so we can make educated design decisions with full awareness of the inherent trade-offs.

The most complete solution would be an n-level I/O cache simulator that could quickly and accurately evaluate many configurations. While there are some advanced CPU cache simulators available [109, 39, 116, 152, 99], storage cache simulators are scarce and lacking. State-of-the-art storage cache simulators are mostly outdated; they either can simulate only a single layer or some fixed set of layers, have limited analysis features, are not easily extendable, or are simply not released to the public [158, 59, 1]. PyMimircache [165] is a popular open-source storage simulator with several useful features that is actively maintained. However, even this simulator is inadequate; it also can simulate only a single layer of cache with no implementation of back-end storage, has no concept of write policy, and its analysis features are limited. The main strength of PyMimircache is its ability to perform MRC analysis on multiple cache replacement algorithms.

It is our position that best practices in cache research need to be broadened to reflect the growing multi-tier configuration space. This chapter makes the following contributions:

1. We explore current trends in cache analysis and propose that best practices in cache research should include the analysis of multi-tier configurations and a more comprehensive set of evaluation metrics (*e.g.*, monetary cost).

2. We describe the critical capabilities of an n-level I/O cache simulator and the initial design of a simulator that we subsequently developed and released publicly [41].

3. We extended PyMimircache to function as a multi-tier cache simulator, experimented with many configurations on a diverse set of real-world traces, and present initial results that support our position.

4.2 Cache Analysis

The fundamental strategy in engineering a cache hierarchy involves placing faster and typically lower-capacity devices in front of slower devices to improve the overall latency of accessing frequently reused data. There is a tangible dollar cost per byte increase when purchasing hardware with better performance attributes. Therefore, it follows that the cache size and speed are closely correlated with the purchase cost. Straightforward logic dictates that performance is constrained by cost, so unless money is in endless supply, the best practice should be to evaluate these metrics together. Surprisingly though, cost is often overlooked during analysis in favor of performance metrics such as raw throughput, latency, or hit/miss ratio [154, 31, 28, 120, 49, 27, 23].

The argument can be made that any improvement in cache performance translates into a reduction in cost when designing a cache, such that the relationship between cost and performance does not necessarily need to be considered. This is situationally true, particularly when evaluating performance in a single-tier caching system. However, in a more realistic, multi-tier storage or CPU cache hierarchy, the large configuration space and complex interactions between tiers produce scenarios where the relative performance per dollar between two configurations is vastly different, necessitating a more complex analysis (see Section 4.4 for examples).

Performance metrics have long been the standard in cache analysis. Recently, additional metrics that are more relevant and informative for specific applications have gained popularity in storage research. The 95^{th} (P95) or 99^{th} (P99) percentile latency, often referred to as *tail latency*, is an important quality of service (QoS) metric for cloud [163, 134] and web [75, 38, 60, 15] services, as well as at the

hardware level [87, 22, 91, 40]. Inter-cache traffic analysis has been used to design more efficient cache hierarchies in modern microprocessors [115]. Reducing the energy consumption of storage systems is beneficial for the environment, lowers operation costs, and promotes advancements in hardware design [24, 89, 127, 146]. Even the total cost of ownership (TCO) can be difficult to calculate when considering all the factors that contribute to capital and operational expenditures (CapEx and OpEx) [90, 89].

It is our position that cache analysis should be conducted using a diverse set of metrics whenever possible. These metrics should be evaluated at various levels of granularity: at each individual layer, some subset of layers, or globally. Moreover, we need to create complex metrics (*e.g.*, throughput/$) that allow for analysis of their informative relationships and reveal critical design trade-offs.

4.3 Multi-Tier Cache Simulation

Simulator Design A general, *n*-level I/O cache simulator with a rich set of features is necessary to thoroughly explore the multi-tier caching configuration space and analyze our proposed metrics. Our initial design considered a broad set of capabilities, including: **(1)** *Write policy* that determines where data is placed upon write requests, including traditional write policies (*e.g.*, write through, write back, write around) and user-defined policies. **(2)** *Admission policy* that controls if and how data is promoted and demoted throughout the hierarchy by request size, address space, or simply whether layers are inclusive or exclusive of each other. **(3)** *Eviction policy* that decides which data to evict when a cache is full and new data needs to be brought in, with support for single-layer or global policies and the ability to add new policies. **(4)** *Trace sampling* techniques (*e.g.*, Miniature Simulations [151]) that reduce the size of a trace to greatly decrease simulation time while maintaining similar cache behavior. **(5)** *Prefetching* to retrieve data before it is requested with techniques like MITHRIL [166] that exploit historical access patterns.

The original design also envisioned an API that would expose data structures at the granularity of individual requests or for any given real timestamp or virtual timestamp (where the trace has only ordered records without their original timing). This would allow users to perform analysis such as examining clean and dirty pages at any level, measure inter-reference recency, calculate stack distance metrics when relevant, or perform any type of analysis offered by our simulation framework on a subset of a trace. We also envisioned coupling the simulator

with modern visualization tools to help users explore the large amount of data it produces. These capabilities defined a broader design direction, while the initial implementation evaluated in this chapter focused on a subset. We continued to develop and use the simulator in subsequent work on knee detection and interactive visualization [42, 43, 142, 173], and later released it publicly as `mtc-sim` [41].

Multi-tier Cache Reconfiguration A major motivation for simulation is seeking optimal cache configurations. However, efficiently reconfiguring a multi-tier cache hierarchy is another challenging problem. In this work, we analyze various physical devices for simplicity, but manually swapping out devices is often not a feasible solution. More likely, multi-tier caches may be dynamically reconfigured in cloud, distributed, and virtual environments, where storage can more easily be allocated through virtualization abstractions. For example, distributed memory caching systems (*e.g.*, Memcached) can greatly benefit from automatically reconfiguring cache nodes in response to changes in workload; but this process can significantly degrade performance as nodes are retired and data is migrated. Hafeez *et al.* developed ElMem, an elastic Memcached system that uses a novel cache-merging algorithm to optimize data migration between nodes during reconfiguration [58]. Moving between configurations in any caching system has a *temporarily* negative impact on performance, until the new caches are fully warmed [176, 26]. Therefore, efficient reconfiguration methods are essential to fully leverage any techniques that find optimal configurations (including simulations).

PyMimircache Extension The initial implementation evaluated in this chapter extended PyMimircache [165], a storage cache simulator with an easily extendable Python front-end and efficient C back-end. We made several simplifying assumptions for this extension and experimented with a subset of the design goals described above. **(1)** We implemented a traditional write-through policy and an "optimistic" write-back policy as global write policies. The write-through policy is consistent and reliable: a block is written to every cache layer and the back-end storage whenever there is a write request. Our write-back policy is optimistic: it only writes to the first layer and assumes this data will be flushed to persistent storage at some point in time, outside of the critical path where it does not affect performance (*i.e.*, we do not account for the write in any other layer). This simplified version of write-back models the best-case performance scenario, which we found useful for exploring the potential effects of write policy. A more re-

alistic write-back would require asynchronous functionality that is not available in PyMimircache, and is a limitation of this work. **(2)** All evicted blocks are discarded rather than demoted (moved or copied) to some lower layer of cache or back-end storage. **(3)** Layers of DRAM are included in our simulations even though we are using block traces, which capture requests for data that was not found in DRAM. This is a limitation of the traces we are using; the simulator itself operates on request identifiers and is not inherently restricted to block traces. **(4)** Throughput is limited by the system where traces were actually captured since we experiment with block traces. For demonstration purposes, we ignore this limitation and assume requests are fed as fast as possible without using the original request timestamps. This allows us to show how we can potentially evaluate throughput when using different hardware configurations. **(5)** We consider each layer to have a portion of its capacity partitioned for caching to emulate various cache sizes at each layer using the specifications of a single device.

A high-level description of how we extended PyMimircache is as follows: **(1)** We feed an original block I/O trace to an instance of PyMimircache. This instance represents the top layer ("L1") of our cache hierarchy. **(2)** This instance generates two output files: i) A log file "L1-log" containing counters for the following: read hits, write hits, read misses, write misses, data read, data written. ii) We specify a new trace file called "L1-trace" which contains read requests that missed in L1, as well as all write requests. As per our assumptions, write requests are not included when using write-back policy and evicted blocks from L1 are never included. These intermediate trace files are stored in memory using Python-based virtual files to avoid disk I/O costs. **(3)** After the L1 instance of PyMimircache completes, we feed the generated "L1-trace" from step 2 as input into another, separate instance of PyMimircache. This emulates our L2 layer. **(4)** We repeat steps 2–3 for L2, L3, etc. **(5)** When all layers have been processed, we aggregate all the log data into a single log file for that experiment. **(6)** We have a higher-level script that we pass parameters to for each layer's device: purchase cost, capacity, and average read and write latencies. This script records and calculates the following metrics for the cache configuration of an experiment: total purchase cost, partitioned device capacities, miss ratio per layer, and total read and write latency incurred. It can be [re]run at any time using previously obtained simulation logs, and is separate from the actual simulation process.

4.4 Evaluation

Workloads In this section, we evaluate simulation results gathered using the Microsoft Research (MSR) traces. These 36 traces, each about a week long, were collected from 36 different volumes on 13 production servers at MSR in Cambridge, Massachusetts, as described in detail by Narayanan *et al.* [111]. The percentage of total requests that access unique blocks (*i.e.*, data used for the first time) in these traces ranges from 1% to 97%, which is representative of the frequency of data reuse. The percentage of total requests that are writes ranges from nearly 0% to almost 100%, making these traces ideal for evaluating the effects of write policies.

We subsequently expanded our experiments to include 9 traces from the Department of Computer Science at Florida International University (FIU) [146] and 106 traces from CloudPhysics [150].

Experimental setup We ran simulations on the MSR traces using between 1 and 3 layers of cache, in addition to the back-end storage device. Each simulation consisted of a configuration of several parameters: cache and back-end sizes, eviction algorithms, and global write policy. The capacity required to hold the entire working set of a trace dictated the cache and back-end storage sizes for every configuration. The back-end size was always fixed to be the same size as the working set, since the data initially resides in the back-end. The cache sizes selected for the first layer of cache are 100 evenly spaced sizes between 1 block (512 bytes) and the size of the working set for that trace. 100 is the default number of points for plotting MRCs with PyMimircache. The second and third layers of cache are 10 evenly spaced sizes within the same range. Using 10 cache sizes for these layers rather than 100 drastically reduced the time required to complete each experiment while still revealing the entire range of metrics (albeit with fewer data points within that range).

In this work, we only present results for configurations using a Least Recently Used (LRU) eviction policy at every layer. We simulated each of the MSR traces using our extension of PyMimircache (see Section 4.3) and then calculated cost and performance metrics using the device specifications described in Table 4.1.

ID	Device	Type	Price	Capacity	Average Latency (Benchmark Source)
D1	G. Skill TridentZ DDR4 3600 MHz C17	DRAM	$150	16GB	0.0585μs r/w (UserBenchmark)
D2	G. Skill TridentZ DDR4 3000 MHz C15	DRAM	$97	16GB	0.0642μs r/w (UserBenchmark) 0.01μs r/w (Vendor)
D3	Corsair Vengeance LPX DDR4 2666 MHz C16	DRAM	$59	16GB	0.0726μs r/w (UserBenchmark)
S2	HP EX920 M.2 NVMe	SSD	$118	1TB	292μs read 1,138μs write (AnandTech) 20μs read 22μs write (Vendor)
H2	WD Black 7200 RPM	HDD	$60	1TB	2,857μs read 12,243μs write (AnandTech)
H3	Toshiba MK7559GSXP	HDD	$65	750GB	17,000μs read 22,600μs write (Tom's HW) 17,550μs read 17,550μs write (Vendor)

Table 4.1: Device specifications and parameters. Each device is denoted with a letter and number for brevity (1 is high-end, 2 is mid-range, and 3 is low-end). Devices S1, S3, and H1 were omitted from the original publication due to space constraints. Prices were obtained from Amazon in September 2019. Benchmarked specifications were compiled from device vendors, AnandTech [7], Tom's Hardware [141], and UserBenchmark [144].

While these comprehensive traces represent a wide variety of workloads, they have only a relatively small working-set size that can easily fit in a modern server's RAM. Therefore, to simulate larger workloads (*e.g.*, bigdata, HPC), we treat the original MSR traces as if they were scaled-down spatial samples of larger traces. We call this technique *reverse-mini-sim*: the reverse of the miniature simulations technique for down-scaling traces introduced by Waldspurger *et al.* [151]. Miniature simulations was shown to be fairly accurate at a sampling rate of 0.001 on the MSR traces, so we multiply the purchase cost (X axis) by a factor of 1,000 times: this simulates a workload whose working set size is 1,000$\times$ larger.

Each data point in our figures represents a configuration with some set of cache sizes. We assume that each layer consists of an independent device with a portion of its capacity partitioned for caching and the remaining capacity as unused. For example, a cyan triangle in Figure 4.1 at Total Scaled Purchase Cost of around $235 represents the average throughput of all requests in a single simulation of the hm-1 trace with an L1 LRU cache of 61,865 blocks partitioned in device D2, an L2 LRU cache of 199,344 blocks partitioned in device S2, and back-end storage of device H3 partitioned to fit the working set of 687,396 blocks.

Cache hierarchy depth Figure 4.1 shows (D1-H3, red) that too little RAM hurts performance but too much wastes money. Adding a bit of SSD cache (D2-S2-H3, cyan) between DRAM and HDD (D2-H3, green) can help, but *not* always (some cyan dots are *below* the green line). Consider the knee of D1-H3 (around X=$500): there are D2-S2-H3 configurations that provide higher throughput for the same cost, same throughput for less cost, and even *both* higher throughput and less cost. Surprisingly, we also see that purchasing more of a cheaper DRAM (D3-H3, blue) for the same cost of a more expensive DRAM (D1-H3) yields overall better performance. Therefore, we can sacrifice DRAM performance for a larger amount of DRAM to get better results.

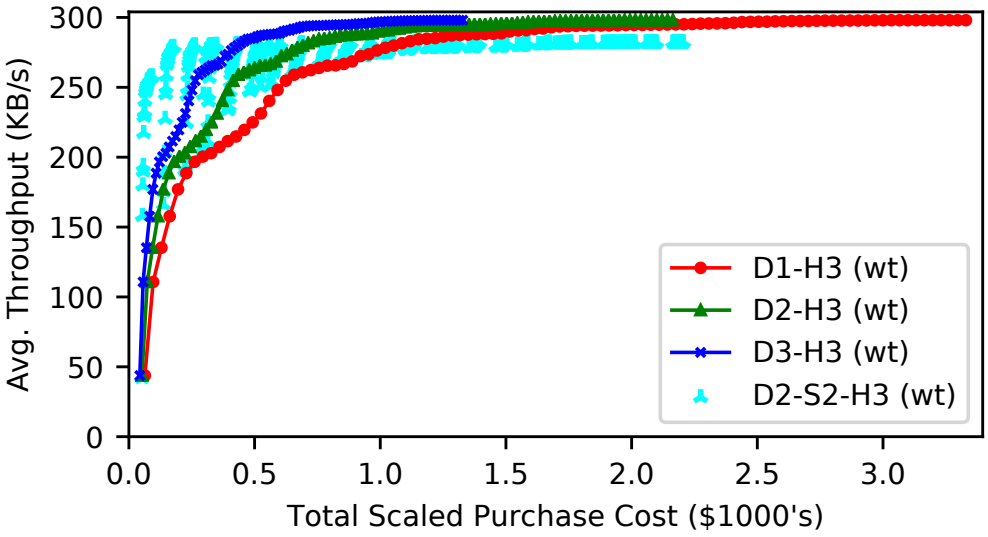


Figure 4.1: Effects of an intermediate SSD tier (Workload MSR hm-1)

Solid-state drive (SSD) degradation Storage devices have an expected lifetime which is typically defined by some amount of I/O. For example, it is well-known that the memory cells within SSDs can only be written to a finite number of times before they are no longer usable [76, 95, 113]. While the lifespan of devices is a parameter that should be considered when estimating the total cost of ownership of a storage system over some period of time, it is also important to evaluate the performance impact this aging process can have. Studies have shown that SSD aging can increase average latency by around 2–3× [78]. To simulate this effect, we multiplied the latency specifications of device S2 and analyzed the results alongside simulations using its original specifications. Figure 4.2 shows that while a new SSD (D1-S2-H2, green triangles) improves performance when inserted into a D1-H2 tier (red), when the SSD is aged (blue and cyan), performance is actually worse than not having the SSD at all. For users with write-heavy workloads or infrastructures where these devices are expected to receive a lot of I/O traffic over a short period of time, choosing to exclude SSDs completely may not only save money, but also yield a similar or better average throughput over time.

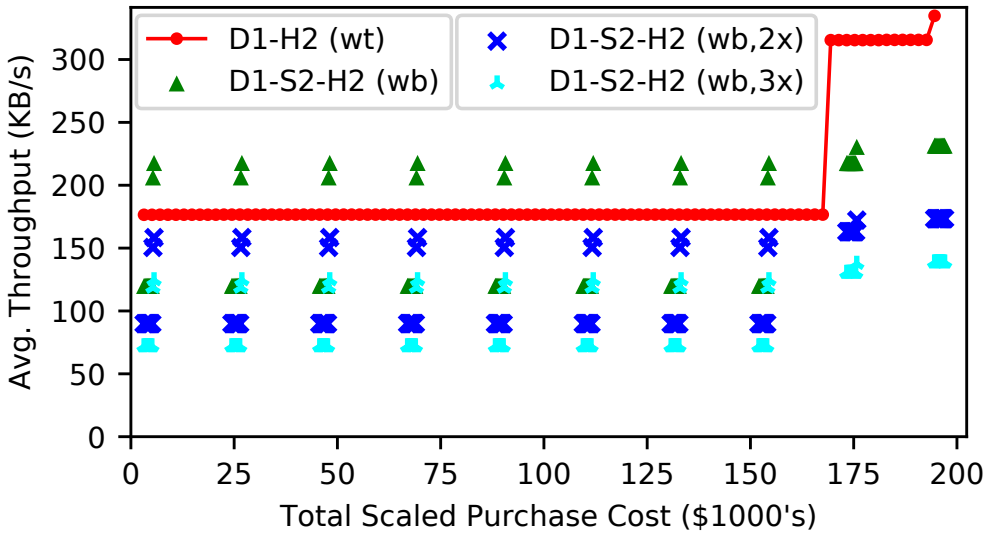


Figure 4.2: SSD Aging Effects (Workload MSR src2-1). 2× and 3× indicate configurations where S2 has 2–3× increased latency due to the potential effects of SSD aging

Device specification variance Storage vendors want to convince consumers that their latest device is competitive. They do so by publishing many device specifications: storage capacity, physical dimensions, hardware interface, durability, energy consumption, and performance metrics. While most specifications are fairly standard, a wide variation of performance metrics can be found, even amongst the same type of device and vendor. Some commonly found metrics are the minimum, average, median, or maximum values for latency, bandwidth, or throughput. These metrics may also be further refined as random or sequential workloads, or separated by reads and writes. These measurements are obtained via benchmarks using some specific workload(s), software environment, and hardware configuration, which are sometimes disclosed at varying levels of detail. This poses a significant problem for consumers, who often are unable to reproduce vendors' performance results. Given such a vast configuration space of variables that can affect performance and the understandable motivation for vendors to publish optimistic results, how can storage devices be reliably compared for their own usage? A handful of independent, reputable websites have emerged by fixing these variables and benchmarking devices from different ven-

dors, and producing realistic, trustworthy specifications: AnandTech [7], Tom's Hardware [141] and UserBenchmark [144].

In this experiment we show the difference between numbers reported by vendors and others. Figure 4.3 shows that inserting an SSD tier between DRAM and HDD provides equal or better performance when using vendor-reported specifications (green and cyan). However, specifications obtained from AnandTech [7] (red and blue) show that the majority of the configurations yield worse average throughput.

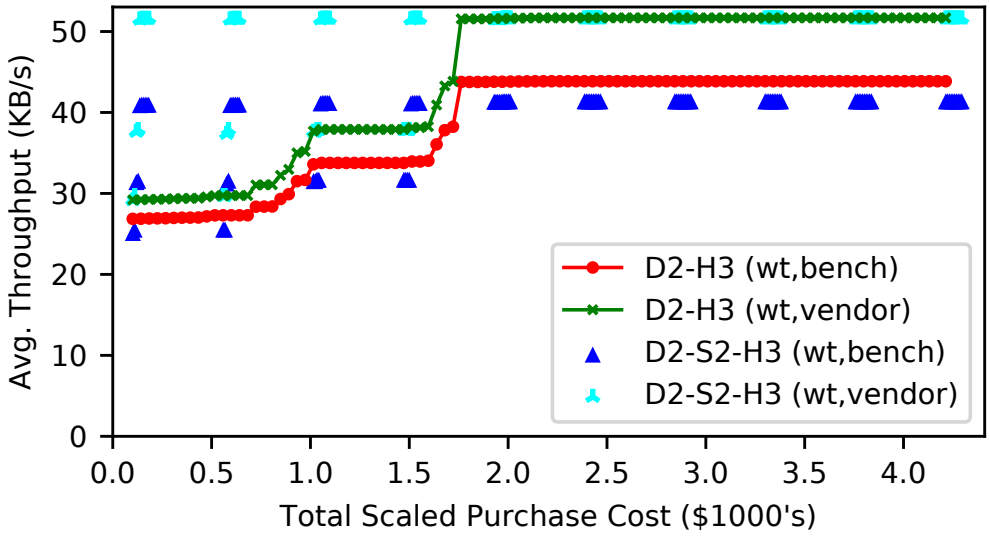


Figure 4.3: Variation between vendor-reported specs and independently operated benchmarks (Workload MSR web-3)

Write Policy The write policy of a cache hierarchy determines how and where data is written whenever there is a write request. Write-through policy ensures data consistency by writing data to every cache and storage device in the hierarchy. However, this incurs the write latency of every device and negatively impacts overall performance. The write-back policy improves performance over write-through by only writing to the cache and then flushing data to back-end storage at a more favorable time. The downside of write-back is that data is at risk of being lost in the event that a cache device fails or the whole system loses power. If reliability is more important, a write-through policy is the obvious choice, but how

much impact will this have on performance? Figure 4.4 compares write-through and write-back policies (policy implementations described in Section 4.3). Using an optimistic write-back (wb) policy we achieve up to 6× better throughput for the same cost as write-through (wt) with the same devices. Note that a more accurate write-back policy will account for the delayed writes, which will tie up the storage devices even during idle times.

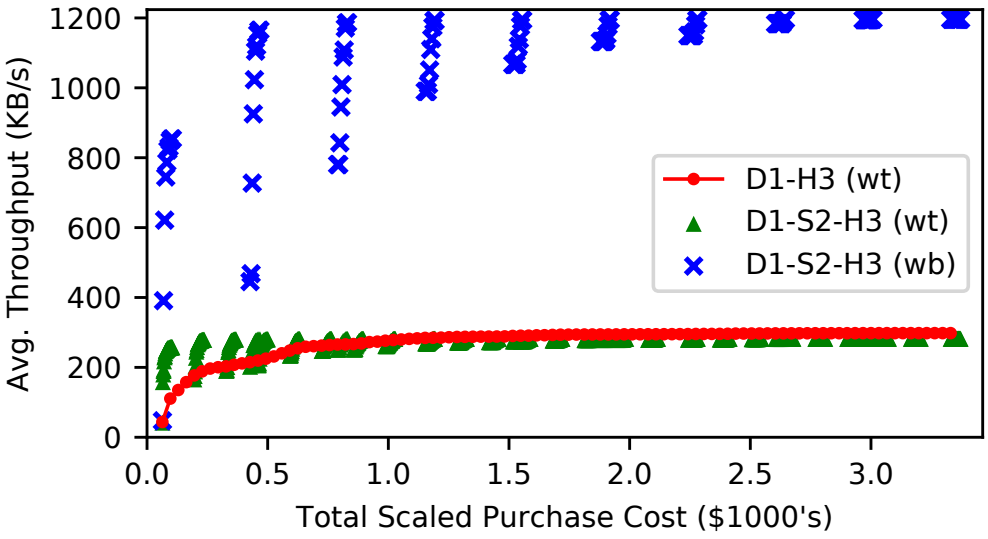


Figure 4.4: Write-through vs. Write-back policy effects (Workload MSR hm-1)

4.5 Chapter Conclusion

Designing and evaluating cache hierarchies has become incredibly complex due to the expanding multi-tier configuration space. In this work, we analyzed the deficiencies of single-tier cache analysis and common cache evaluation metrics. We propose that best practices in cache research should include the analysis of multi-tier systems, as well as the evaluation of a more comprehensive set of metrics (particularly monetary cost) and their relationships. We developed an *n*-level I/O cache simulator that models multi-tier cache hierarchies with an arbitrary number of levels and supports analysis across a broad range of configurations and metrics. The initial implementation evaluated in this chapter extended PyMimircache and

was used to experiment with a wide variety of workloads. We presented interesting and counter-intuitive results that demonstrate the need for multi-tier simulation and analysis. We continued to develop and use the simulator in later studies and released it publicly as `mtc-sim` [41].

Chapter 5

Accelerating Multi-Tier Storage Cache Simulations Using Knee Detection

Storage cache hierarchies include diverse topologies, assorted parameters and policies, and devices with varied performance characteristics. Simulation enables efficient exploration of their configuration space while avoiding expensive physical experiments. Miss Ratio Curves (MRCs) efficiently characterize the performance of a cache over a range of cache sizes, revealing "key points" for cache simulation, such as knees in the curve that immediately follow sharp cliffs. Unfortunately, there are no automated techniques for efficiently finding key points in MRCs, and the cross-application of existing knee-detection algorithms yields inaccurate results.

We present a multi-stage framework that identifies key points in *any* MRC, for both stack-based (*e.g.*, LRU) and more sophisticated eviction algorithms (*e.g.*, ARC). Our approach quickly locates candidates using efficient hash-based sampling, curve simplification, knee detection, and novel post-processing filters. We introduce *Z-Method*, a new multi-knee detection algorithm that employs statistical outlier detection to choose promising points robustly and efficiently.

We evaluated our framework against seven other knee-detection algorithms, identifying key points in multi-tier MRCs with both ARC and LRU policies for 106 diverse real-world workloads. Compared to naïve approaches, our framework reduced the total number of points needed to accurately identify the best two-tier cache hierarchies by an average factor of approximately $5.5\times$ for ARC and $7.7\times$ for LRU.

We also show how our framework can be used to seed the initial population for evolutionary algorithms. We ran 32,616 experiments requiring over three million cache simulations, on 151 samples, from three datasets, using a diverse set of population initialization techniques, evolutionary algorithms, knee-detection algorithms, cache replacement algorithms, and stopping criteria. Our results showed an overall acceleration rate of 34% across all configurations.

5.1 Introduction

A cache's miss ratio is one of the most important predictors of its performance. A miss-ratio curve (MRC) for a given cache and replacement algorithm plots the cumulative miss ratio for all accesses as a function of the cache size, providing a powerful tool for analyzing the performance of live systems and dynamically adjusting cache configurations as workload conditions change [151, 15]. MRCs can also inform offline evaluations such as comparing caching algorithms or analyzing monetary cost vs. storage-system performance [44].

There are many efficient techniques for generating MRCs [102, 135, 159, 150, 137, 66, 151, 49]. MRC-reported miss ratios are good indicators of expected performance (*e.g.*, throughput), but real system performance can vary due to additional factors including device characteristics, write policies, and admission policies [44]. Alas, repeatedly reconfiguring and testing a real caching system, with all possible cache sizes, is prohibitively expensive due to the slowness of storage I/O.

Since experimenting with physical devices is costly and time-consuming, simulation offers a more practical way to explore this large search space and evaluate trade-offs such as latency vs. cost. A common first step is to sample a workload: approximation algorithms enable accurate simulation of cache behavior using only a fraction of the original trace data. Small sampled traces can then be used to construct an MRC accurately, enabling quick evaluation of cache performance [150, 151]. Many storage-cache simulators have been developed that replay traces while attempting to faithfully reproduce real system behavior [1, 97, 165]. However, even simulations can be too expensive to allow exploring a large number of configurations or optimizing live systems in real time. For example, consider a cache with a maximum size of 100 GB. Simulating every 1 GB size step would require 100 experiments. In a multi-tier setup, the number of simulations grows with the number of tiers; a two-tier configuration would require 100^2 experiments, three tiers would need 100^3, and so on. Thus, it is essential to

explore this vast configuration space efficiently.

Creating an MRC requires a sequence of cache references. In a multi-tier cache, references to level $n + 1$ come from misses in—and write evictions and flushes from—level n; thus the MRC for $n + 1$ directly depends on the cache size chosen for level n. A naïve exploration of *multi-tier* configurations would require a separate simulation for each point in level n's MRC to identify misses that become references at level $n + 1$, and hence to compute the level $n + 1$ MRC. Since an MRC may contain hundreds of points (one for each potential cache size), this approach quickly becomes intractable. Thus, a crucial second step for evaluating multi-tier caches is to limit the number of simulations by intelligently selecting the cache sizes to evaluate at each level.

Intuitively, the most promising candidates are points where a little extra cache space produces a relatively large drop in the miss ratio; such points are often visible as "knees" in MRCs—*e.g.*, points A, C, and D in Figure 5.1. (Note that although B has sharp curvature, it is not of interest since C provides a much lower miss rate.) Given enough computational resources, we may be interested in also selecting some points in the large gradually sloping regions that cover a significant range of cache sizes. We refer to both types of points as *key points* from here on.

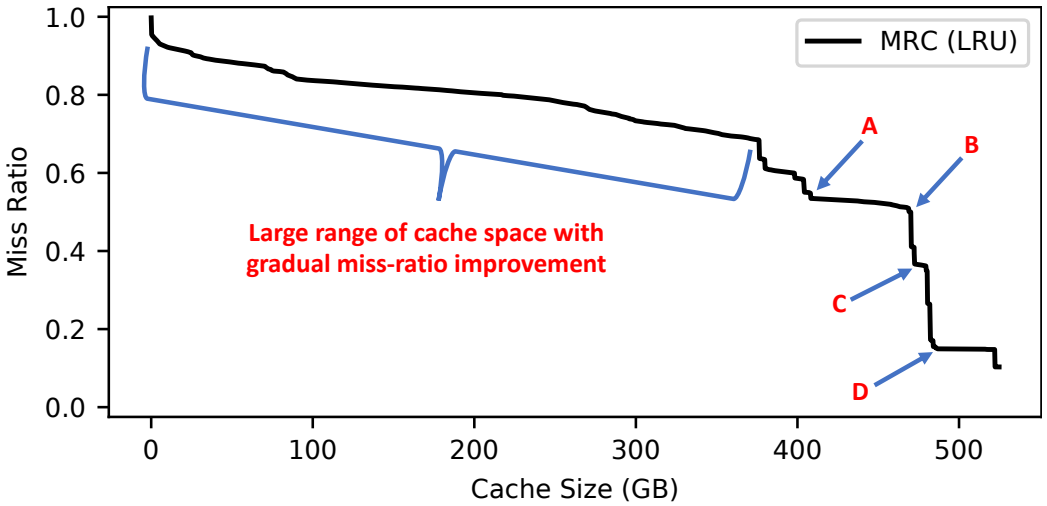


Figure 5.1: MRC for trace w10, annotated to illustrate several key points: useful "knees" (points A, C, and D), a useless "cliff" (B), and a large range of cache sizes with relatively gradual miss-ratio improvement.

In this chapter, we describe a multi-stage framework designed to pick an appropriate yet small number of key points in MRCs: **(1)** We first approximate the

MRC accurately using a hash-based sampling technique [150, 151]; **(2)** Next, we use the Ramer-Douglas-Peucker (RDP) line simplification algorithm [121] to reduce noise by eliminating minor variations in the curve; **(3)** We then run a multi-knee detection algorithm on the remaining points to find cache sizes that provide the greatest miss-ratio improvement for the lowest cost. Our framework currently implements eight different knee-detection algorithms, including our novel Z-Method and modified, multi-knee versions of five widely used single-knee detection algorithms [9, 126, 124, 139]; and **(4)** Finally, in post-processing we remove less interesting points, add points in gradually-sloped regions (if desired), and then select the final points based on a ranking that uses hierarchical clustering and relevance metrics.

Building on our previous work [42], we additionally provide a more in-depth analysis of our Z-Method algorithm and demonstrate how our framework can accelerate the optimization of multi-tier caching systems using evolutionary algorithms. This chapter makes several contributions:

1. We establish the novel methodology of using multi-knee detection to efficiently identify optimal multi-tier cache configurations;

2. We present a framework that combines several techniques to find a minimal number of key points in MRCs for both stack and non-stack caching algorithms;

3. We introduce Z-Method, a new multi-knee detection algorithm that uses statistical outlier detection;

4. We demonstrate that, compared to naïve approaches, our framework significantly reduces the number of 2-tier cache evaluations needed to identify good configurations by a factor of $5.5\times$ for ARC and $7.7\times$ for LRU;

5. We evaluate our framework for the additional application of seeding the initial population of evolutionary algorithms. Our results show an overall acceleration rate of 34% across a highly diverse set of configurations and datasets; and

6. We release the code library containing all techniques used in this work [42].

The next section provides some background on MRCs, knee-detection algorithms, and MRC cliff removal techniques. Section 5.3 presents the point-selection techniques used in our framework, leading to the design of the Z-Method

algorithm in Section 5.4. We evaluate all of our techniques' ability to find key points in MRCs in Section 5.5. We then present an additional evaluation of how our framework can be used to optimize multi-tier caching systems using evolutionary algorithms in Section 5.6. Finally, we summarize our conclusions and highlights in Section 5.7.

5.2 Background

5.2.1 Miss Ratio Curves (MRCs)

A key feature of some MRCs is their monotonicity. Cache replacement algorithms such as LRU are stack-based, which means they satisfy the *cache-inclusion property:* the content of a cache of size n is always a subset of a cache of size $n+1$. Ultimately, this property ensures that the miss ratio will never degrade as we increase the size of the cache, producing a monotonically decreasing curve. However, more sophisticated algorithms such as ARC [103] are not stack-based and thus the inclusion property does not hold, causing them to produce MRCs that may contain both convex and concave regions [151]. Thus, non-stack-based MRCs need not be strictly decreasing.

5.2.2 Knee-Detection Algorithms

Many heuristic algorithms have been developed that find a single knee in a curve, although the precise definition of a "knee" varies. One can define a knee point as the point with the maximum curvature in a function. For continuous functions, curvature [52] is mathematically defined as follows:

$$K_{f(x)} = \frac{f''(x)}{(1 + f'(x)^2)^{\frac{3}{2}}} \tag{5.1}$$

However, knee-detection algorithms are applied to discrete sets of points, instead of a well-defined continuous function. As such, there are several methods to measure the curvature of the discrete sequence. Menger curvature [139, 126] defines the curvature for a sequence of three points as the curvature of the circle circumscribed by those points. This method relies only on a local criterion, using only three points to estimate the knee point without considering any others. As such, noisy data can lead to poor accuracy when estimating the knee point. We use this method as a baseline reference to compare with other methods.

The L-method [124] fits two straight lines from the head of a curve to a candidate point, and from the candidate point to the curve's tail. The candidate that minimizes the Root Mean Squared Error (RMSE) between the straight lines and the points of the curve is returned as the knee point; this represents the sharpest angle in the curve.

Similar to the L-method, Dynamic First Derivative Thresholding (DFDT) [9, 10] tries to identify the point where the function has a sharp angle. Instead of fitting two straight lines, this method relies on the first derivative of the curve. After computing that derivative, a thresholding algorithm is used to identify the value that separates the derivative values as "high" or "low." The knee is then the point with a derivative value that is closest to the previously computed threshold.

Kneedle [126] uses the point on the curve that is furthest away from a line defined by the head and tail points of the curve. Both axes of the original curve are normalized to $[0, 1]$ to easily find the point with maximum curvature. Kneedle was designed for single or multi-knee detection in a streaming scenario where new data is arriving continuously. The authors used this technique to detect relevant points for network congestion control and latency.

There are several algorithms that can find multiple knee points in a curve, but they have limitations that make them unsuitable for MRCs. The Kneedle algorithm's primary use case is anomaly detection, where it serves as an initial filter to reduce the number of candidates needing further analysis. As such, for Kneedle, recall is more important than precision: it aggressively captures all anomalies, producing many false positives. In some cases it is possible to reduce the number of false positives, but doing so requires extensive tuning of its sensitivity parameter.

A few multi-knee detection algorithms have been developed for use in multi-objective optimization problems, where the notion of a knee guides the exploration of meaningful candidate solutions [169]. However, these problems use a stricter definition of a knee that assumes a set of well-behaved, Pareto-optimal points. Several other knee-detection methods [124, 9, 10, 139, 8] are only effective at finding a *single* knee in a small and relatively smooth set of points. In contrast, MRCs can consist of a relatively large number of points, can be noisy or non-monotonic, and commonly contain more than one significant knee. In this work, we had to develop techniques to overcome these limitations (see Section 5.3) by enabling these algorithms to find multiple knee points.

5.2.3 Cliff Removal Techniques

An alternative to detecting knees in an MRC is to modify the underlying cache-replacement policy so that it does not have any cliffs, yielding a *convex* MRC. Talus [14] removes cache performance cliffs by dividing the cache into two *shadow partitions*, each receiving a fraction of the input load. Varying the sizes and input loads of each partition emulates the behavior of smaller or larger caches. Given an MRC as input, Talus computes the partition sizes and their respective input fractions to ensure that their combined aggregate miss ratio lies on the convex hull of the original MRC. Originally proposed for processor caches, Talus inspired the SLIDE [151] technique for removing performance cliffs from software caches that employ sophisticated non-LRU replacement policies. CliffHanger [32] applied a similar idea to key-value web caches, but instead estimated the MRC gradient without explicitly constructing one.

The recent eMRC [94] technique generalized Talus's cliff removal to multi-dimensional *miss ratio functions*, such as the three-dimensional miss-ratio surface for a two-tier cache. The eMRC convex-hull approximation technique leverages the absence of cliffs to efficiently generate the miss ratio function for a multi-tier cache. However, eMRC *requires* convexity, which limits its applicability to modeling multi-tier cache systems that employ cliff removal. As real-world multi-tier cache systems do not yet perform cliff removal, eMRC is unable to approximate their non-convex MRCs. In contrast, our approach does not require convexity to accelerate multi-tier cache evaluations, making it broadly applicable to production deployments of existing caches.

5.2.4 Evolutionary Algorithms: Population Initialization

The initial population of an evolutionary algorithm functions as the first guess at a set of good solutions to an optimization problem. The quality of this first set can significantly influence the quality of the final solution and the speed at which an algorithm converges [3, 79]. Studies have shown that some evolutionary algorithms, such as Genetic Algorithms, are more sensitive to the initial population, while other algorithms like Particle Swarm Optimization are less dependent on the initial population [48]. This sensitivity has also been shown to be problem-dependent, such that an algorithm may be influenced by the initial population for certain functions, numbers of dimensions, or population sizes [3].

There are several categories of population-initialization techniques. Common stochastic variants, such as random initialization, are favored for their simplicity,

generic nature, and applicability to a wide range of problems [98]. They are also popular because they produce a relatively uniform distribution as the population size increases. But statistical methods such as Latin Hypercube sampling [110] and quasi-random sequences (*e.g.*, Halton sequence [145]) have been shown to outperform random initialization for most problems [12].

There are also application-specific techniques aimed at specific, narrow problems. They often exploit domain knowledge or use a problem's characteristics with specific evolutionary algorithms. These methods have been applied to improve convergence speed in problems such as grammar-guided genetic programming [50] and flexible job-shop scheduling [170]. Initialization techniques in this class typically perform better than more generic variants, but they are usually limited to specific problems.

Lastly, there are domain-agnostic, general heuristics applicable to problems that meet some set of conditions. Examples include approaches developed to optimize any two-stage stochastic mixed-integer problem [140]. Such techniques can be applied to any problem where some of the variables are constrained to integer values.

The techniques described in this chapter focus on a domain-agnostic, generic approach that can be applied to any problem where a curve that is correlated to the solution can be derived from features of the input data. We show how such techniques can be used on miss ratio curves to optimize a cache with evolutionary algorithms.

5.3 Point Selection Techniques

In this section, we introduce our framework for finding multiple key points in MRCs. We first designed a pre-processing stage to deal with the large volume of data (Section 5.3.1). Next, we made substantial modifications to each point-selection technique, enabling them to output a set of multiple knees instead of just one (Section 5.3.2). Finally, we added a post-processing stage (Section 5.3.3) that filters and ranks knees based on an appropriate definition.

5.3.1 Pre-Processing

A curve can contain an arbitrary number of data points. The largest MRC that we evaluated contains 276K points even after sampling-based size reduction [151];

the original MRC is 10,000× larger. However, the knee-detection algorithms evaluated in this work were originally designed to work with small or partial data, such as for clustering optimizations. Our main idea is to reduce the number of points while preserving those that follow the shape of the curve; this greatly reduces the computational costs of subsequent steps while also improving knee-detection accuracy by minimizing irrelevant fine-grained variation.

The Ramer-Douglas-Peucker (RDP) algorithm modifies a curve by finding a similar one with fewer points [121]. RDP fits a line between the curve's endpoints and then finds the point in between that is farthest from this line. If the distance between that point and the line is over a given threshold, the curve is split there and the algorithm is reapplied recursively on the two new segments. Once the distance is smaller than the threshold, *all* intermediate points are removed. The main drawback of RDP is the need to define a threshold, which can be understood as the maximum allowed reconstruction error. The choice of threshold is difficult because it depends on the curve's complexity.

We modified the original RDP algorithm to address this difficulty. Instead of defining a threshold for the maximum allowed perpendicular distance between a point and the fitted straight line, we use a relevance-based cost metric that computes the difference between the fitted straight line and the data points in the current segment.

We evaluated four different metrics that assess how far our linear reduction is from the original data: Root Mean Squared Log Error (RMSLE), Root Mean Squared Percentage Error (RMSPE), Relative Percent Difference (RPD), and symmetric mean absolute percentage error (SMAPE). Of these four, the best performance came from SMAPE: it found the smallest set of points that minimized the reconstruction error.

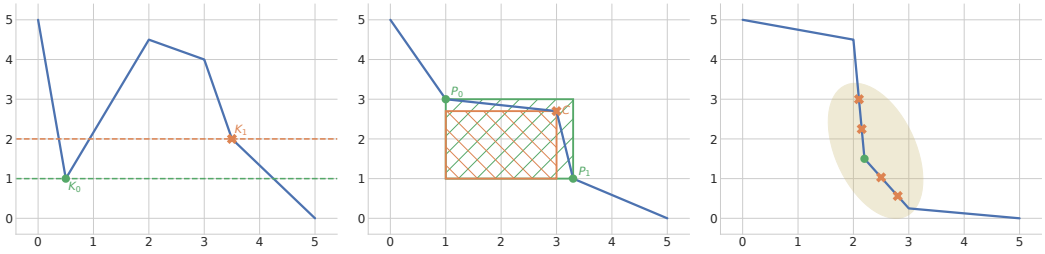
5.3.2 Methods

Except for Kneedle, the algorithms we evaluate in this work (see Section 5.2.2) were not designed to detect multiple knees. Thus, we developed a recursive algorithm that can be used to adapt any single-knee detection technique to handle multiple knees. The basic idea is to use a single-knee technique to select the best knee in a segment. We then split the current segment at that knee, and for each new segment check whether it is sufficiently linear (computed using the SMAPE metric). If not, we repeat the process recursively. Apart from applying this recursive generalization, we do not alter the core knee-detection technique, using it as a black box. All of the methods we evaluated, even Kneedle, require our pre- and

post-processing methods to work properly on MRCs.

5.3.3 Post-Processing

Given the differences between single- and multi-knee detection and the large number of points produced by using our recursive strategy on some of the knee-detection algorithms, we developed three different filters to further reduce and select the most relevant knees.



(a) Undesirable knee filter. (b) Cliff-point removal filter. (c) Knee clustering and ranking.

Figure 5.2: Graphical representation of the post-processing methods. (a) Representation of the filter that removes unwanted knees. Knee K_1 is removed since knee K_0 achieved better performance. (b) Representation of the overlapping rectangles used by the corner detection algorithm. Point C is identified as a cliff point and then removed. (c) Representation of the clustering and ranking elements. The orange ellipse represents the cluster of knee candidates. The orange candidates are filtered out, while the green knee is selected as the best representative based on Equation 5.2.

The first filter, shown in Figure 5.2a, removes useless knees. When dealing with non-monotonic curves, a knee-detection algorithm can incorrectly choose a knee that is *above* a previously detected one. We remove such knees since they are sub-optimal and do not add useful information.

The second filter, shown in Figure 5.2b, removes cliff points located after a smooth, near-horizontal area that precedes a sharp descent. These points are found using a corner-detection algorithm that computes the overlapping area of two rectangles. The first rectangle, shown in green, is drawn from the neighboring points P_0 and P_1 (assuming that RDP pre-processing was used, these are the previous and following points) that are adjacent to the knee candidate point C. The second rectangle, drawn in orange, has its corners placed at C and the lower

left of the green rectangle. The filter computes the percentage overlap between these two rectangles, and a knee candidate is removed if the overlap exceeds a threshold.

The third and final filter, shown in Figure 5.2c, uses a hierarchical clustering algorithm to group knees by their distance along the x-axis, using a percentage of the x range as a threshold. After grouping the knees into clusters, the knees within each cluster are ranked based on their relevance score, computed from two metrics: (i) the improvement given by each knee (*i.e.*, how much it decreases on the y-axis from the highest knee in the cluster) and (ii) the smoothness of the improvement, computed using the coefficient of determination (R^2). Specifically, the relevance score S is given by Equation (5.2):

$$S(K_i, \vec{L}) = |K_h - K_i| \cdot R^2(\vec{L}), \tag{5.2}$$

where K_i is the i^{th} knee, and K_h is the knee with the highest value on the y-axis (in a single cluster). $\vec{L}$ is a vector containing all the knees in the cluster up to and including the i^{th} one: $\vec{L} = [K_0, ..., K_i]$. The highest-ranked knee in each cluster is selected as its representative knee.

5.4 Z-Method

5.4.1 Design Concepts

Our design for Z-Method was inspired by the DFDT [9] and DSDT [10] knee-detection algorithms, which use first and second derivatives, respectively. In statistics, a *z-score* (also known as a *standard score*) is a transformation that normalizes a data value by quantifying how many standard deviations away it is from the mean; typically, a point whose z-score has an absolute value greater than three is considered an outlier [2]. For the purpose of detecting knees, such outliers in the second derivative indicate a significant change in the y-axis. The foundation of our Z-Method technique is in detecting such outliers and intelligently selecting knees among them.

Although the second derivative is useful, we found that large and small knees often tend to cluster, causing several points in close proximity to be selected, rather than the single most optimal knee in the vicinity. To remedy this, we introduced two hyper-parameters, dx and dy, that specify the minimum x and y distances, respectively, between all selected points. These parameters limit the

total number of knees selected and give users control over the algorithm. For example, users interested only in large knees can give relatively high values for dx and dy to minimize the number of points.

Z-Method was designed to function independently of the techniques described in Sections 5.3.1 and 5.3.3. As such, it works for curves that are non-monotonic, with both convex and concave regions (see Section 5.2.1).

5.4.2 Algorithm Description

Algorithm 1: Z-Method Multi-Knee Detection

Input: Data D with (x, y) points, dx, dy, dz
Output: List of (x, y) points corresponding to knees

$\Delta x \leftarrow \text{length}(D) \cdot (dx/100)$
$\Delta y \leftarrow (\max(y) - \min(y)) \cdot (dy$ / 100)
$D'' \leftarrow$ calculate second derivative of D
$Z \leftarrow$ calculate z-scores for D''
$K \leftarrow$ empty list
$zLimit \leftarrow 3$ # standard outlier threshold [2]
while *TRUE* do
    $C \leftarrow$ points in Z with z-score $\geq zLimit$
        and at least Δx and Δy apart from all points in K
    $Z \leftarrow Z$ - C
    if $zLimit \leq 0$ ***and*** $\text{length}(C) == 0$ **then**
        Remove points from K to ensure that y always decreases as x increases
        return K
    $G \leftarrow$ group all points in C such that all adjacent points in each group are $< \Delta x$ apart
    sort G in descending order by max z-score of each group
    foreach $group$ *in* G **do**
        $p \leftarrow$ point in $group$ with the lowest y value
        if p is at least Δy from all points in K **then**
            $K.append(p)$
    $zLimit \leftarrow zLimit - dz$

As shown in Algorithm 1, Z-Method takes as input a discrete curve *D* consisting of an ordered list of (x, y) points, along with parameters dx, dy, and dz. The parameters dx and dy both influence the size and number of selected knees, while dz controls the maximum number of iterations in the main selection loop (lines 7–20).

We first convert dx and dy, specified as percentages, into absolute values Δx and Δy for the input curve (lines 1–2). This normalization ensures that these parameters function similarly for different curves. We then approximate a list of second derivatives of the curve, D'', using second-order polynomial fitting [25]; next we calculate the z-scores of all points in D'' as Z, both of which are found in linear time (lines 3–4). We initialize a list K to contain all selected knees, and set our starting value of $zLimit$ to 3, since a z-score ≥ 3 is a widely accepted value for outliers [2] (lines 5–6).

We then enter the main selection loop (lines 7–20), which selects points and progressively decrements the $zLimit$ value. First, we create a new list C that contains candidate points: points in Z that have a z-score greater than the current $zLimit$ and are at least a minimum Δx and Δy distance from all other already-selected points (lines 8–9). The complexity of this step is $O(|C| \times |K|)$. All candidate points C are removed from Z so that we will not consider them again in future iterations (line 10). The termination clause is then checked (lines 11–13) to ensure that we have candidate points to operate on.

We next group the candidate points C into G, such that the adjacent points in each group are less than Δx apart, based on the dx parameter constraint (line 14). This takes $O(|C|)$ time, effectively forming groups of points such that there is at least Δx distance between every group. We then sort the groups in G in descending order by the maximum z-score of each group (line 15). Here, we are sorting the location of each group in the list of groups G rather than the points within each group. From each group, we select the point with the lowest y value (line 17); we then check that the selected point is not within a minimum Δy distance from other points that have already been selected, enforcing the dy parameter (line 18). The complexity of this loop is $O(|G| \times |K|)$. A point is added to the list of knees K if it satisfies this constraint (line 19). We then decrement the $zLimit$ by dz and continue with the next loop iteration (line 20).

This loop terminates only after we have reached a $zLimit \leq 0$ and there are no remaining points that can be selected given the dx and dy parameters (line 11). At $zLimit = 0$, we consider all points in D that have not already been considered in previous iterations. By starting at z-score ≥ 3 and iteratively approaching 0, we select the largest knees first and gradually lower our threshold for how big a knee

should be.

Finally, we eliminate any points that may have been poorly selected due to non-monotonicity in the curve. A final pass removes points where increasing the x value makes the y value worse (line 12); in our MRC application, such points are clearly undesirable. This simple pass requires time linear in the size of K. The overall complexity of this algorithm is therefore $\mathcal{O}(|D| \times |K|)$, where D is the size of the input curve and K is often a trivially small value. For example, with dx set to 5%, enforcing at least 5% distance on the x-axis between each selected knee point, the maximum size of K would be 20.

5.4.3 Parameters

We present qualitative evaluations of the algorithm's parameters dx and dy, as well as its overall success at finding *key points*. Furthermore, we demonstrate that Z-Method is effective for both stack and non-stack algorithms, by evaluating with LRU and ARC cache replacement policies.

Parameter: *dx* The dx parameter has several functions within Z-Method. It is provided as a percentage of the maximum cache size in the given MRC. The most transparent effect of dx is that it constrains the minimum x distance, or cache size, between selected points. Since no two points can have an x distance less than dx between them, this provides an upper bound on the total number of selected points, and also influences the number of points that are actually selected. Because it affects the "grouping" stage of the algorithm, dx also effectively defines the width of the knees.

Figure 5.3a shows the effects of dx on workload w09 with LRU cache replacement, with dy fixed and dx set to 1%, 5%, and 10%. The black line in each plot represents the MRC for LRU cache replacement. The green dots are the points selected by Z-Method when using the dx and dy parameters indicated in the legend. The vertical orange lines show the second derivative z-score of the MRC at each cache size. Because the z-score values have a large, workload-dependent dynamic range, we truncate them at 10 in this plot and for the remainder of this chapter. A z-score range up to 10 is sufficient to identify all points considered as outliers (*e.g.*, z-score ≥ 3).

For the MRC plotted in Figure 5.3a, we will focus on the knee(s) in the region of cache sizes between approximately 425 GB and 475 GB. In the top plot with $dx = 1\%$, Z-Method considers this region to contain four separate knees, since

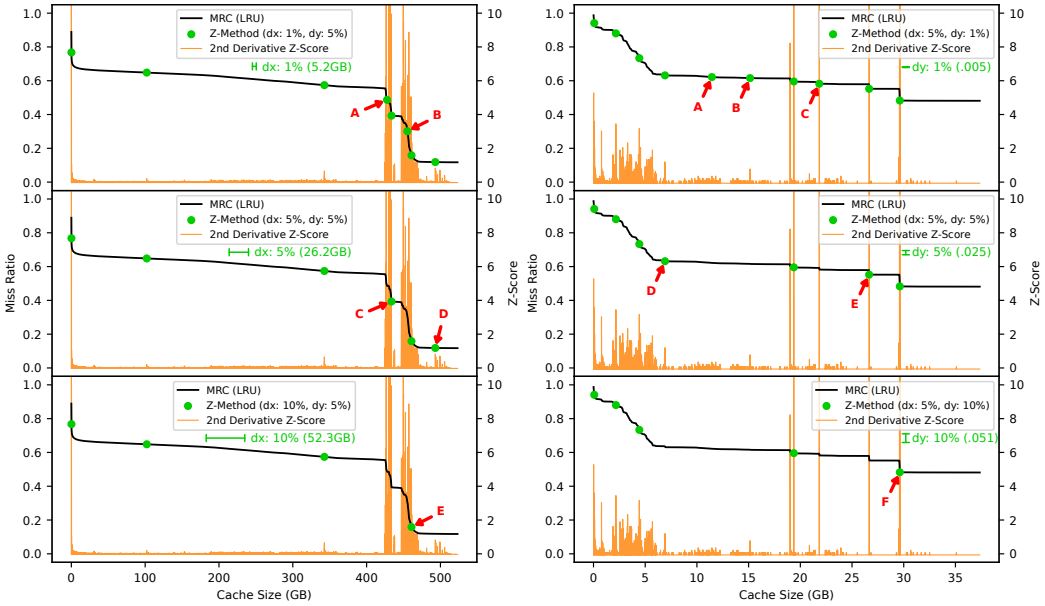
their distances from each other are at least 1% of the maximum cache size. When we move from 1% to 5% in the middle plot, we can see that points A and B from the top plot have been removed. Those points are now within dx of each other, so they are grouped together; we are now left with two points at wider, more prominent, knees.

A similar effect is seen when we increase dx from 5% to 10% in the bottom plot. The two knees at points C and E are grouped together and C is removed. Point D is also removed, since its cache size is less than 10% away from point E. Significantly, the knee point E was favored rather than the less interesting point D.

Parameter: *dy* The dy parameter is also specified as a relative percentage, which is then converted into an absolute value for the given MRC. It functions similarly to dx, except that it constrains the y distance, or delta miss ratio, between any two selected points. This effectively influences the height of knees and how many points are selected, while providing an upper bound on the total number of points that can be selected.

Figure 5.3b shows the effects of dy using workload w62 with LRU cache replacement by fixing dx and varying dy between 1%, 5%, and 10%. The format is otherwise the same as in Figure 5.3a. In the top and middle plots, the most interesting change occurs at point C. With $dy = 1\%$ in the top plot, this very small knee is considered significant and is selected. However, when we increase dy from 1% to 5% in the middle plot, points A, B, and C are removed, as the y distance between these points and adjacent points is now less than dy. Similarly, point E is removed when we move from 5% to 10% in the bottom plot, while the taller knee point F is retained. We can also see that point D is removed as well, as increasing dy reduces the number of selected points.

It is important to note that for both of these parameters, we are not guaranteed to *always* have a point that is dx or dy apart from every other point. Enforcing a hard separation rule would add a great deal of complexity and would provide little benefit, since we already select points by their order of importance.



(a) Effects of Z-Method parameter dx. (b) Effects of Z-Method parameter dy.

Figure 5.3: (a) Effects of dx = 1% (top panel), 5%, and 10% on trace w09. Red arrows denote points in a panel that were not selected when the dx value increased (next panel down). (b) Effects of dy = 1% (top panel), 5%, and 10% on trace w62. Red arrows denote points in a panel that were not selected when the dy value increased (next panel down). The minimum distance that Z-Method enforces between points is shown below the legends in green.

Parameter: *dz* The dz parameter controls the amount that the $zLimit$ variable is decremented in each iteration of Z-Method. This affects the overall running time by influencing the total number of iterations. It can also affect the size of candidate point groups. For example, with a starting $zLimit$ of 3, $dz = 0.1$ would only consider points with a z-score ≥ 2.9 on the second iteration, but $dz = 0.5$ would consider a potentially larger set of points that have a z-score ≥ 2.5. While this may seem significant, the dx and dy parameters are still the predominant influence on how groups are formed, so we did not observe any trends or significant changes when modifying dz.

Finding key points In Figure 5.4, we show the points selected by Z-Method with dx and dy set to 5%, for multiple workloads using both LRU and ARC cache replacement policies. We evaluated these plots based on whether or not they selected all of the points that we consider *key points*. To reiterate, Z-Method should first select the largest knee points and then eventually select those within any regions that cover at least 5% of the x and y axes.

The first row of plots (LRU1-3) shows examples where Z-Method performed well for LRU. All prominent knees were selected and large ranges of cache space with gradual decreases in miss ratio also contained an adequate number of points. The second row of plots (LRU4-6) shows examples where Z-Method missed key points. For example, in plot LRU4, points A and B missed the knee points directly to their left. There were similar issues in LRU5 and LRU6.

We show how to improve the lower-quality points A and B in LRU4 by modifying Z-Method parameters. The green points were selected using the default dx of 5%, while the blue points were selected using a dx of 3.2%. These blue points more accurately capture these two knees, and are more optimal than A and B.

There were also cases where Z-Method could pick slightly less optimal points due to extreme shapes in a curve and the nature of the z-score metric. This can be observed in plot LRU5, which exhibits a nearly flat region followed by a massive, steep knee, then another nearly flat region. Point C is not quite at the bottom of the knee because the z-score at the very bottom was slightly below the standard threshold for an outlier of 3. This could be remedied by lowering the threshold (and modifying Z-Method's parameters if needed). It should be noted that Z-Method will still pick a point that is relatively close to the knee in these edge cases.

The third row of plots (ARC1-3) shows where Z-Method performed well for ARC. In addition to always selecting prominent knees and points in gradually

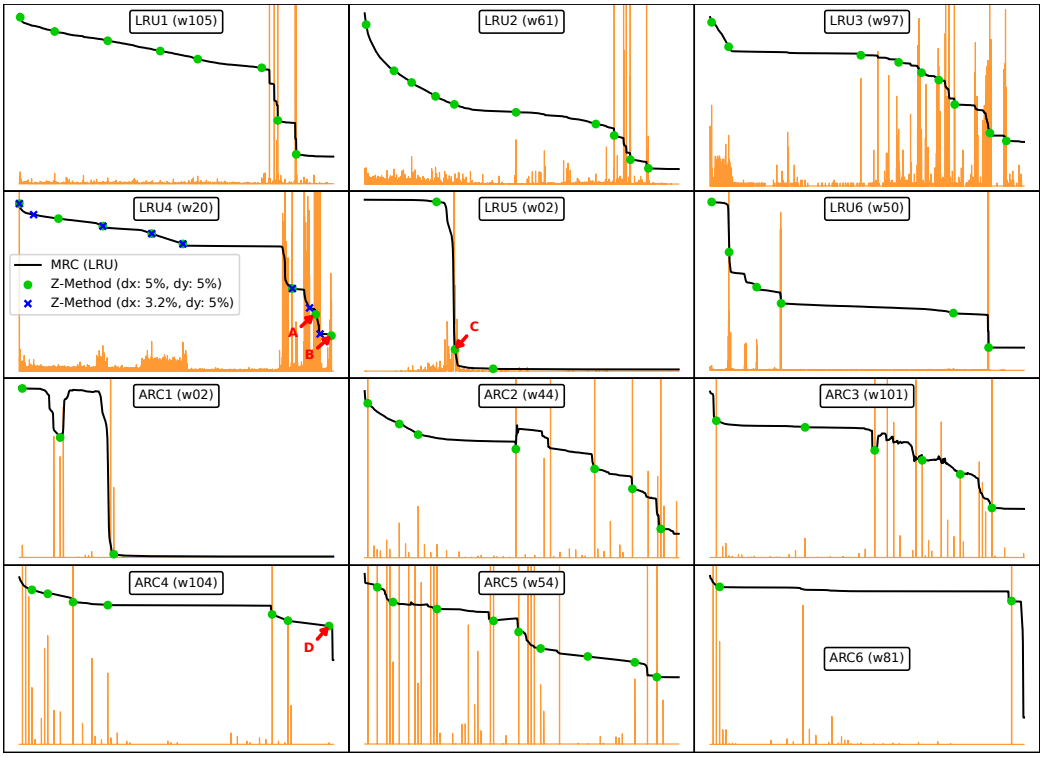


Figure 5.4: From the top: the first row shows LRU plots where Z-Method picked fairly good points; the second row has LRU plots where Z-method missed a few, better points. The third and fourth rows are the same but for ARC (third row good points selected; fourth row missed some points). The plot labeled LRU4 shows points selected by Z-Method using a dx of both 5% (green) and 3.2% (blue). Sub-optimal points A and B in LRU4 were selected using dx of 5%, missing nearby knee points. The knees were appropriately selected when dx was lowered to 3.2%. Sub-optimal point C in LRU5 was selected due to the extreme shape of the curve and the standard z-score threshold of 3. The knee was appropriately selected when lowering the threshold. Sub-optimal point D in ARC4 was selected due to the ARC MRC being generated with too few points. The knee point to the immediate right was selected when the number of points in the MRC was increased from 100 to 220.

sloped regions, we also see that points were never selected in concave regions where the miss ratio increased due to the non-monotonicity of ARC. A key feature of Z-Method is that it will never select points with a higher miss ratio than any other previously selected points with a lower cache size.

The fourth row of plots (ARC4-6) depicts a situation where Z-Method missed key points. In cases such as ARC5, modifying the parameters was sufficient for identifying higher quality points, but plots ARC4 and ARC6 exhibit a problem that is unique to non-stack-based cache eviction algorithms (*i.e.*, ARC). Unlike stack-based algorithms (*e.g.*, LRU), we cannot easily generate a fine-grained MRC that includes every potential cache size. Instead, best practice is to sample the workload [151] and then generate the MRC using a subset of points that still preserves the shape of the curve. This is typically done using 100 points. We used 100 points to generate all ARC MRCs during the point selection process throughout this work, but plotted the z-score and points selected against MRCs generated using 1000 points to better show how Z-Method selects points in MRCs that more closely represent the "true" curves. This value worked well for the majority of our workloads, but there were edge cases (*e.g.*, ARC4 and ARC6) where the z-score did not accurately capture important knees present at the very end of the curve. For example, point D in ARC4 was selected because there was a very small knee with a positive z-score at the top of the cliff, but there was a much larger knee to its immediate right. This could be remedied by increasing the number of points used to generate the MRC. 220 points were enough to remedy this issue for both ARC4 and ARC6 (not shown in the plots due to complexity). The task of generating an MRC is separate from Z-Method, so we did not include this value as a parameter.

In all cases where Z-Method missed key points, there were slight modifications that enabled those points to be selected. We fixed dx and dy to 5%, the z-score outlier threshold to 3, and generated ARC MRCs using 100 points for this evaluation, since we do not yet have a way of automatically selecting the ideal values. Even so, Z-Method still found all key points in the overwhelming majority of the MRCs we examined with these values.

5.5 Evaluation: Miss Ratio Curves

In this section, we evaluate our framework's ability to find key points in miss ratio curves. We first compared the accuracy of 8 different knee-detection algorithms, including Z-Method, for identifying knees in single-tier MRCs, and then evaluated

our framework's ability to quickly find optimal multi-tier configurations.

5.5.1 Experimental Setup

We evaluated our techniques on 106 real-world block traces collected by CloudPhysics [150], each representing a week of virtual disk activity from production VMware environments. We used hash-based spatial sampling [150, 151], with a size-based sampling rate ranging from 0.1 to 0.0001, to reduce these workloads and thus the running time while maintaining an accurate representation of the originals. We dynamically varied the rate by powers of 10, such that each sampled trace was guaranteed to contain between 100K and 1M requests. The traces contain heterogeneous request sizes, so we also transformed all requests into 4 KB block-aligned operations to facilitate accurate sampling, consistent with previous work [94].

To evaluate multi-tier systems, we extended PyMimircache [165], a cache simulator with an easily modifiable Python front end and an efficient C back end. Our extension generates two-tier MRCs by simulating an L1 cache with the original sampled trace, then simulating L2 with the requests that missed in L1. L1 cache sizes were selected using the MRC of the original trace, while L2 sizes were chosen using the MRCs of each intermediate trace. The total miss ratio of the two-tier configuration was calculated as the product of the miss ratios of L1 and L2. We modeled a simple write-back policy by treating both reads and writes as cache references, as done in previous work [94]. The cache eviction policy was configured as either LRU or ARC and was the same in both tiers. We generated two-tier MRCs for each trace, for both LRU and ARC replacement policies, resulting in a total of 212 MRCs.

5.5.2 Knee-Detection Algorithms

We evaluated the accuracy of our framework using Z-Method and several other knee-detection algorithms: Curvature, DFDT, Kneedle, L-Method, and Menger. We also included the *Fusion* method, which considers all points retained by RDP and relies on our post-processing filters to select relevant knees.

Kneedle finds knees using peak-detection methods, and can be used for single-knee detection by selecting only the highest possible peak; we call that approach *Kneedle Recursion*. We analyzed each method's ability to find knee points that had been manually curated in the 212 single-tier MRCs by four domain experts.

Most of our techniques have one or more hyper-parameters that can influence which points are selected. To achieve the best performance for each MRC, it is necessary to tune the hyper-parameters of each algorithm appropriately. While the default parameters offered acceptable performance, a more complete evaluation requires optimized parameters [21]. Therefore, we developed a cost function and ran an optimization algorithm for each knee-detection method using all 212 MRCs.

When designing the cost function, we carefully considered the target use case of our framework. We want to find the ideal parameters that have the lowest error globally across all MRCs, while keeping the number of knees as close as possible to the number of knees identified manually. Constraining the number of knees is necessary since our framework is designed to pick only the most relevant points. A technique that finds parameters that correctly identify all knee points but also selects many non-relevant points, while having a high recall score, would be inefficient for our use case.

We use the Matthews correlation coefficient (MCC) [29] as the basis of our cost function. MCC measures classification quality by considering the balance ratios of the four confusion matrix categories: true positives and negatives, and false positives and negatives. Although the knee-detection problem is better modeled as a regression, we based our evaluation on binary classification, since we wanted to control the impact of false positives and negatives (*i.e.*, non-relevant points being classified as knees and vice-versa). Prior work [29] has shown that the MCC is more informative than an F1-score for evaluating accuracy in binary classification problems. As such, the cost function we used was the following:

$$\mathit{Cost}(E, K) = \frac{1}{n}\sum_{i=0}^{n} \mathit{MCC}(E_i, K_i) + \max\left(\left(\frac{1}{n}\sum_{i=0}^{n}|K_i|\right) - K_t, 0\right), \quad (5.3)$$

where E represents the expected (manually selected) knees for all MRCs, and K represents knees picked for all MRCs (n defines the number of MRCs) by our framework using some configuration of hyper-parameters and a knee-detection algorithm. $\mathit{MCC}(E_i, K_i)$ represents the MCC computed from the expected and detected knees of the i^{th} MRC. Finally, $|K_i|$ represents the number of knees detected in the i^{th} MRC, and K_t represents a threshold for the acceptable number of knees.

Figure 5.5 shows our evaluation. Three techniques stand out: Fusion, Kneedle, and Z-Method. Fusion achieves tighter margins than all other techniques, spanning only 0.23 MCC between the upper and lower quartiles, suggesting that

the expected performance in unseen traces would be well-bounded. Kneedle and Z-Method achieve the highest median MCC values of 0.45 and 0.50, respectively, with Z-Method having a smaller standard deviation when compared with Kneedle. The much tighter bounds of Z-Method are more significant than the improvement in median, making Z-Method the ideal candidate for our multi-tier evaluation.

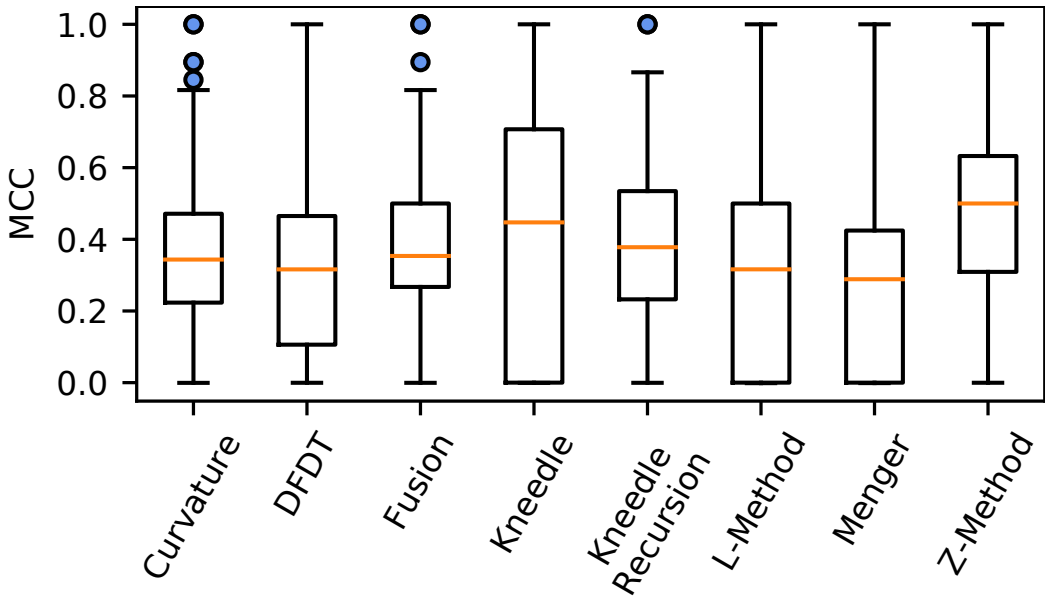


Figure 5.5: An MCC evaluation of 8 knee detection algorithms using our optimized hyper-parameters for accurately identifying knees that were manually selected by experts. Higher MCC values and lower standard deviations are better. Kneedle and Z-Method have the highest median MCC values of 0.45 and 0.5, respectively, with Z-Method achieving much tighter bounds.

We also experimentally measured the time and memory usage of our framework using each knee-detection algorithm for all MRCs. We used the default hyper-parameters for each algorithm, as they do not significantly impact the overheads. The results of the running-time benchmarks are shown in Figure 5.6.

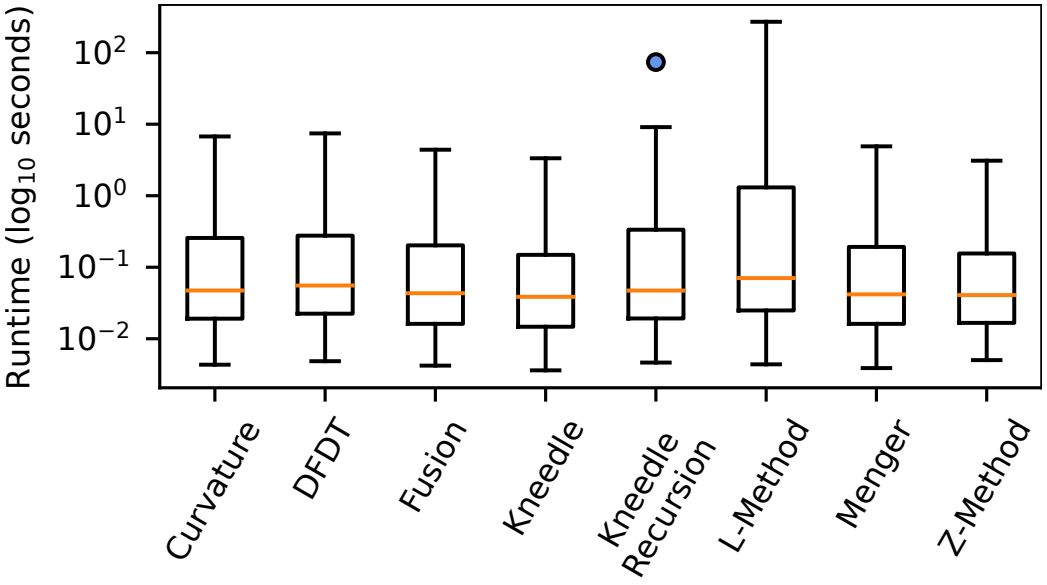


Figure 5.6: Running times for 8 knee-detection algorithms. The y-axis scale is logarithmic. L-Method displays a considerably higher running time than other methods, due to its high complexity. Other methods are comparable, with Kneedle and Z-Method having the lowest median running times.

All of the algorithms have comparable execution times across all MRCs with the exception of L-method. This is especially true for the upper quartile of L-Method, which is 1307.79 ms. The second highest upper quartile is Kneedle Recursion, which is 74.4% lower than L-Method at 334.62 ms. This is expected since L-method uses straight-line fitting ($O(n^2)$) for each point to detect the ideal knee, leading to a time complexity of $O(n^3)$. Combined with our recursive algorithm that enables it to detect multiple knees, the expected time complexity for L-method is $O(n^3 \log n)$. Note that Kneedle (non-recursive version) and Z-Method have the lowest median running times of 38.59 ms and 40.81 ms, respectively.

The memory overhead is nearly linear in the file size for each MRC, ranging from approximately 53 KB to 71 MB after sampling. There were no significant differences across any of the techniques, so we do not present any further memory overhead analysis.

5.5.3 Multi-Tier MRCs

Miss-ratio curves are typically used to find configurations that minimize both miss ratio and cache size(s). We seek multiple configurations that are optimal in two or more objectives.

Consider designing a multi-tier cache, with many device options, for a large workload. With an unlimited budget, one could simply purchase enough DRAM to hold the entire data set, but that is rarely economical. Instead, most system administrators will want to trade cost off against performance, meaning that they will be interested in *Pareto-optimal* solutions, *i.e.*, those where a given objective cannot be improved without making one or more others worse.

Only a subset of all possible cache configurations are Pareto-optimal. When a set contains every Pareto-optimal configuration for a given workload and no others, it is called the *true Pareto-optimal front*. Any point in this front minimizes the cache size(s) and the miss ratio; the front as a whole can be considered to mark the "best" points.

However, it is often not feasible to find the true Pareto front for a large configuration space. Instead, the space can be sampled in an attempt to find optimal points, creating a *Pareto approximation*. Our work aims to find a minimal number of key points in MRCs. Thus, we are trying to find the most significant Pareto-optimal points by efficiently generating accurate Pareto approximations of multi-tier MRCs.

There are multiple metrics for evaluating the quality of Pareto approximations [88]; the most commonly used is the *HyperVolume Indicator (HVI)* [20],

which measures the size of the space between the points in a front and a user-defined reference point; a larger space is better.

Figure 5.7 shows an example of how HVI is measured in a 3-dimensional space. The blue shape represents a simple linear series descending from (0,0,10) to (10,10,0). If this were a two-tier MRC, the x-axis would be the L1 size, y-axis the L2 size, and the z-axis the miss ratio. The hypervolume is the volume between points on the Pareto front (here, the blue shape) and a user-defined *reference point*, here the *nadir point*[1] at (10,10,10), where all objectives are maximized. To find the hypervolume of the point at (5,5,5), we draw a rectangular prism from it to the reference point. The resulting $5 \times 5 \times 5$ cube has a hypervolume of 125. If we were to instead find the hypervolume of the point (4,4,4), we would have a $6 \times 6 \times 6$ cube with a hypervolume of 216. Thus, configurations with lower cache sizes and miss ratios result in larger hypervolumes. The total hypervolume of a dataset is the non-overlapping hypervolume of all points on its Pareto front, making HVI a useful metric for our multi-knee detection framework.

[1] Although the reference point is placed at the largest coordinates, prior literature on hypervolume indicators uses the term "nadir" rather than "zenith" because it represents the worst performance; we follow that convention.

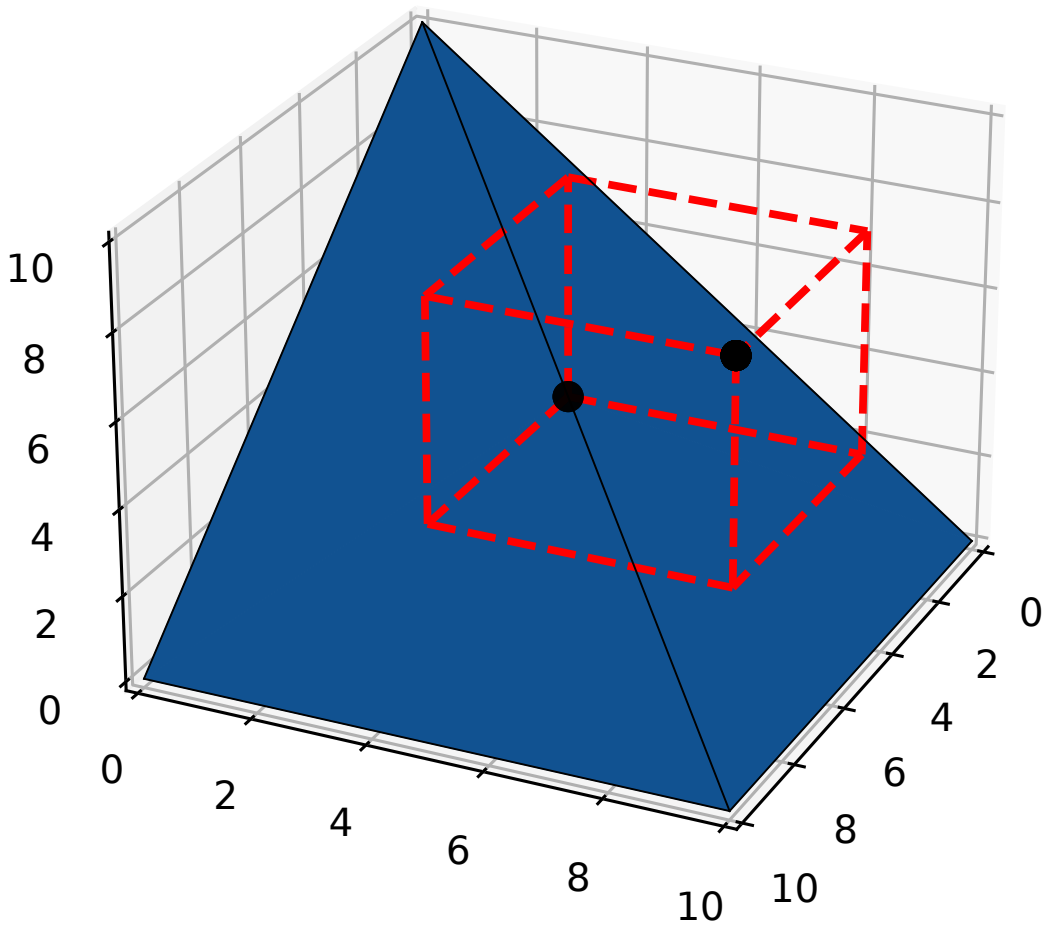


Figure 5.7: An example of how the *HyperVolume Indicator (HVI)* is calculated for a single point of a 3-dimensional data series. A cube is drawn from the reference point (10,10,10) to the data point (5,5,5) creating a $5 \times 5 \times 5$ cube with a hypervolume of 125.

Another metric highly relevant to our problem is the Ratio of Non-Dominated Individuals (RNI) [136], which is the fraction of dataset points that are on the Pareto front. As discussed earlier, points not on the front represent sub-optimal configurations, so a higher ratio is better. RNI does not measure the magnitude of quality; instead, it informs us of a point selection technique's efficiency. Therefore, evaluating HVI and RNI together is a comprehensive approach to analyzing techniques that find the minimal number of key points in MRCs.

We evaluated our framework across all 212 two-tier MRCs using Z-Method, compared to a naïve approach of selecting evenly-spaced points. We also tried geometrically-spaced points, but this yielded worse results than even spacing so we omit them from this analysis. It was not practical to evaluate every point in MRCs containing thousands of points, so we used 50 evenly-spaced points (Even50) as a reasonable approximation of the full configuration space and the true Pareto front. We varied the number of evenly-spaced points to most closely match Z-Method's average HVI or number of points, resulting in Even4, 10, and 13.

In Table 5.1, we show the averages across all 212 MRCs of the number of points selected, HVI as a percentage of Even50's HVI, and RNI. When measuring the efficiency of a method, a lower number of points and a higher RNI are better; when measuring the accuracy of a method, higher HVI is better. The number of points for even spacing is always constant, calculated as $X + X^2$ for two-tier MRCs using Even*X*. Z-Method has HVI similar to that of Even10 for ARC and to Even13 for LRU, but Z-Method evaluates $5.5\times$ fewer points for ARC and $7.7\times$ for LRU to get those results. This efficiency is also reflected in Z-Method's RNI of 0.94 for ARC and 0.97 for LRU. Conversely, the RNI of the evenly-spaced methods ranges from 0.28 to 0.41, meaning that the majority of points they select are sub-optimal and uninteresting to explore.

Method	Avg. Points		Avg. HVI %		Avg. RNI	
	ARC	LRU	ARC	LRU	ARC	LRU
Even4	20	20	59.63	61.00	0.30	0.28
Even10	110	110	86.43	83.31	0.39	0.40
Even13	182	182	90.41	91.07	0.41	0.34
Even50	2550	2550	100.00	100.00	0.33	0.34
Z-Method	23.33	20.33	86.99	90.75	0.94	0.97

Table 5.1: Evaluation results of our framework using Z-Method across 2-tier ARC and LRU MRCs, derived from 106 real-world block traces collected from CloudPhysics. The averages of 3 metrics are presented for each algorithm: number of points (lower is better), HyperVolume (HVI) as a percentage of Even50's HyperVolume (higher is better), and Ratio of Non-Dominated Individuals (RNI) (higher is better).

In Figure 5.8, we show box plots for all 212 MRCs. Figure 5.8a displays the number of points selected by each technique. We can see that Z-Method and Even4 selected approximately the same median number of points. A significant result is that in the worst case, Z-Method still picked fewer points than Even10. We can also see cases where Z-Method picked very few points. There are times when such a low number of points is appropriate, but this can also represent cases where the default parameters were too conservative, resulting in too few points and a low HVI.

Figure 5.8b displays the HVI as a percentage of Even50's HVI. This figure reveals several outliers where Z-Method performed poorly, but also many cases where it had a higher HVI than Even50. These results inform us about Z-Method's sensitivity to its hyper-parameters. The default parameters worked well for the majority of our workloads, but needed to be tuned better for others. With the right parameters, Z-Method performed better than naïve approaches while selecting a minimal number of points. Lastly, Figure 5.8c displays the RNI. Z-Method consistently had a greater RNI than all of the evenly spaced methods, indicating that it properly identified key points. We can also see that there were diminishing returns when increasing the number of evenly spaced points. The median RNI decreased from Even13 to Even50, meaning that Even50 selected many points that did not contribute to the Pareto front.

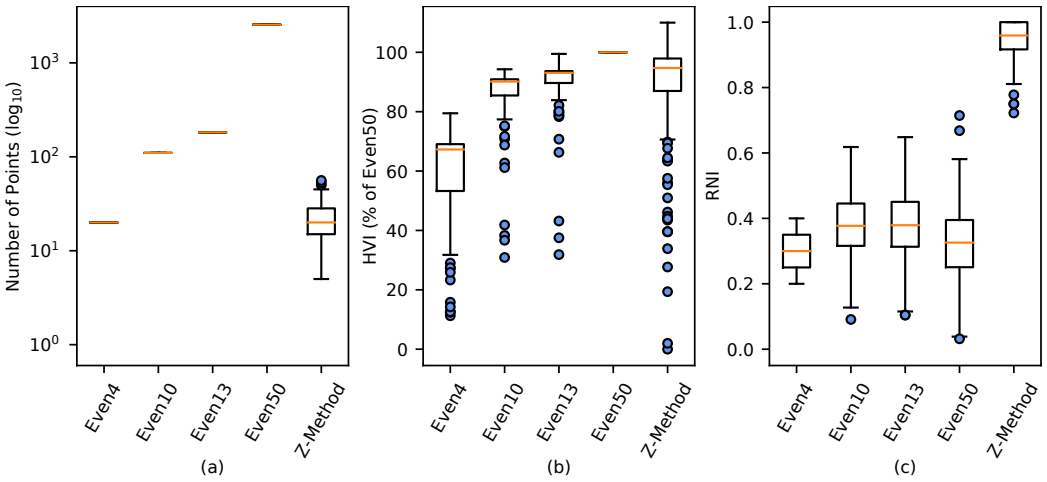


Figure 5.8: Evaluation results of our framework using Z-Method across 2-tier ARC and LRU MRCs, derived from 106 real-world block traces collected from CloudPhysics. Box plots of 3 metrics are presented: (a) number of points (lower is better), (b) HyperVolume (HVI) as a percentage of Even50's HyperVolume (higher is better), and (c) Ratio of Non-Dominated Individuals (RNI) (higher is better).

In Figure 5.9, we show visualizations of the points chosen by each method for a few selected two-tier MRCs with fairly different characteristics.[2]

[2] A similar figure that appeared as Figure 4 in our earlier work [42] inadvertently showed visualizations for Even5 instead of Even4.

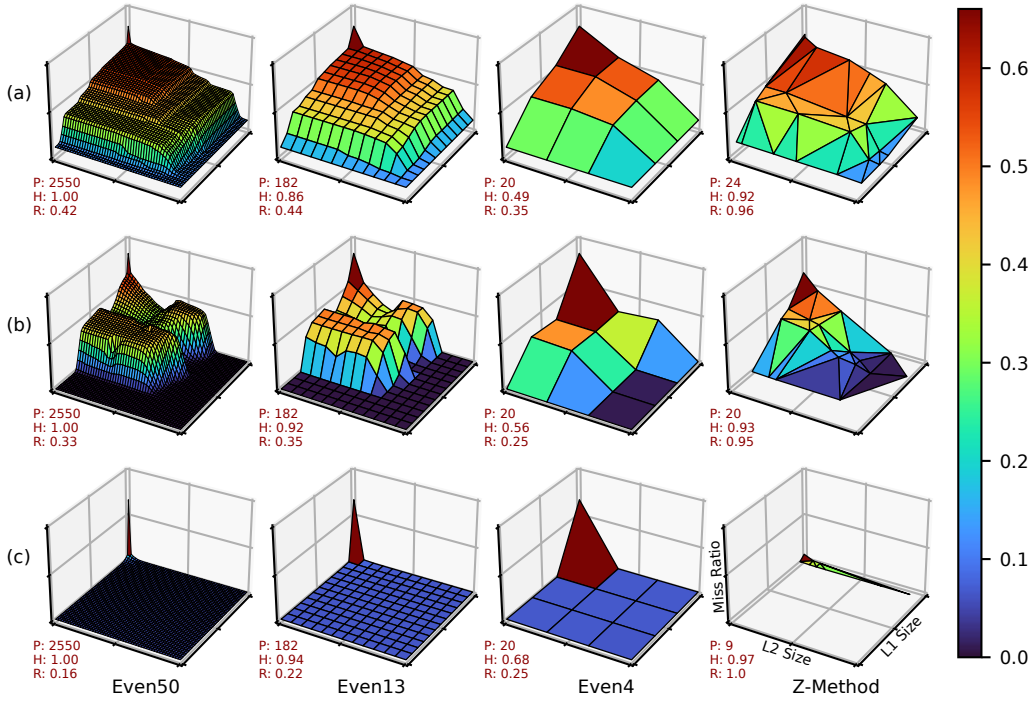


Figure 5.9: Examples of point selection on two-tier MRCs that highlight three different commonly observed scenarios. Each point represents the total miss ratio of a configuration of some L1 and L2 sizes. The x and y axes are the normalized L1 and L2 sizes, respectively, while the z-axis is the miss ratio. Axis labels are omitted to reduce clutter. Each row contains MRCs of a single workload using 4 different point-selection methods, listed at the bottom of each column. The P value indicates the total number of points (lower is better), H is the HyperVolume as a percentage of Even50 (higher is better), and R is the Ratio of Non-Dominated Individuals (higher is better).

Figure 5.9a (top row) displays the MRCs for workload *w04* using LRU replacement, where several knees of various sizes are followed by gradually-sloped regions. We can see that Z-Method accurately selects each knee, achieving 92% of Even50's HVI while evaluating over $100\times$ *fewer* points. Conversely, Even13 and Even4 perform poorly, selecting points at the tops of the cliffs before the knees, resulting in lower HVI values of 86% and 49%, respectively. When several knees are present, Z-Method has more opportunities to exploit these significant improvements in miss ratio, performing much better than evenly-spaced points.

Figure 5.9b (middle row) displays the MRCs of workload *w66* using ARC replacement, which exhibits large amounts of non-monotonicity, creating several hilly regions. Z-Method finds the interesting knee points at the hill bottoms, while the post-processing filter prevents selecting any points at the hilltops. Z-Method is even more efficient here than in the previous figure while still being highly accurate, selecting only 20 points and achieving 93% of Even50's HVI. Even13 gets close to Z-Method's HVI, but requires $9.1\times$ more points.

Finally, Figure 5.9c (bottom row) displays the MRCs of workload *w06* using ARC replacement, which contains only a couple of interesting points at the very beginning of the plot. Z-Method finds 3 points in this tiny space that are more optimal than those found by Even4 or Even13; it also does not waste time exploring the large, flat MRC region that offers almost no improvement in miss ratio. With only 9 points, Z-Method achieves 97% of Even50's HVI, while Even13 evaluates $20.2\times$ *more* points but achieves only 94% of Even50's HVI. MRCs that contain only a handful of good points are fairly common, even in multi-tier settings, and our framework dramatically reduces the time spent exploring them.

5.6 Evaluation: Population Initialization

In this section, we show how our multi-tier knee detection framework can also be applied to population initialization for evolutionary algorithms, to search large configuration spaces more efficiently [51, 37]. In many cases, evaluating the fitness of a configuration is an expensive operation, making the speed of convergence particularly important. The initial population of an evolutionary algorithm functions as the first guess at a set of good solutions, so the population's quality can significantly influence the quality of the final solution and the speed at which an algorithm converges [3, 79]. As such, heuristics to intelligently select a population have been developed for a variety of scenarios and optimization problems [145, 12]. Evaluating multi-tier caching systems fits this scenario well; re-

playing a workload repeatedly on numerous cache configurations can be costly in both time and money. We demonstrate how the key points found by our multi-knee detection framework can be used to seed the initial population of evolutionary algorithms.

5.6.1 Experimental Setup

We configured each experiment with choices for an input I/O trace, a population-initialization technique, an evolutionary algorithm, a knee-detection algorithm, a cache-replacement algorithm, and two parameters controlling the stopping criteria for the optimization. For each configuration, we analyzed the convergence of an evolutionary algorithm with each of the population-initialization techniques and with our multi-knee detection framework.

We used our PyMimircache [165] cache simulator extension (see Section 5.5.1) for all experiments. Simulation enabled us to study a wide variety of configurations, as trace replay on real hardware would be far too slow and would limit the configuration space we could explore. We optimized a single variable, cache size, for the performance metric of I/O operations per second (IOPS) per dollar ($), or IOPS/$. We calculated theoretical values for the IOPS using the same methodology as eMRC [94], and computed dollar costs from a given configuration's cache size and current market values for that type of device [7, 141]. We normalized both the IOPS and dollar cost and then combined them to determine IOPS/$.

We evaluated this use case on three different sets of real-world block traces obtained from CloudPhysics [150] and the publicly available FIU [147] and MSR [112] traces, for a total of 151 individual traces. We used uniform randomized spatial sampling [150, 151] with a size-based sampling rate R (ranging from 0.1 to 0.0001) on the larger traces to reduce the running time while maintaining an accurate representation of the original trace. Our sampling produced a fairly diverse set of MRC sizes, with a mean of $70{,}446 \pm 110{,}014$ blocks, ranging from 263 to 829,424 blocks.

We evaluated the speed of convergence of evolutionary algorithms using four population-initialization techniques: our multi-knee detection framework, random initialization, Latin Hypercube sampling (LHS) [110], and Halton sequences [145]. For techniques that include randomization (all but multi-knee), we ran them three times with different random seeds to obtain stable results. Three seeds are generally considered the minimum acceptable number for this type of analysis. However, given the size of our dataset and configuration space, even three random

seeds resulted in experiments that required significant running time while still remaining viable. We ran a total of over **3M** experiments, which sufficiently covers the search space, allowing us to evaluate our proposed solution with statistical confidence. We experimented with three types of evolutionary algorithms: Generalized Differential Evolution 3 (GDE3) [83], a Genetic Algorithm (GA) [65], and Particle Swarm Optimization (PSO) [80]. We focused on a subset of the knee-detection algorithms available in our framework: Menger, Kneedle, and Z-Method. We selected these three because Menger represents a baseline knee-detection method that uses a local feature, Kneedle is a well-known algorithm that greatly benefits from our framework, and Z-Method is our novel algorithm designed for this specific application.

We used both Adaptive Cache Replacement (ARC) and Least-Recently Used (LRU) cache replacement policies. These two popular policies present interesting scenarios for multi-knee detection since the MRCs produced by LRU are guaranteed to be monotonically decreasing, while ARC's MRCs can contain both convex and concave regions (see Section 5.2.1). Lastly, we enforced two types of stopping criteria for the optimization: (1) the number of evaluations and (2) an objective value that was some percentage of the "best" value for that configuration. The number-of-evaluations stopping criterion was fixed at 300. We found this value sufficient to allow approximately 99% of our experiments to converge. To handle the objective-based stopping criterion, for each trace we simulated 1,000 cache sizes evenly spaced from the minimum to the maximum and calculated their IOPS and dollar costs. We obtained the maximum IOPS/$ from these simulations, and treated it as the "best" value when calculating the objective stopping criterion for all optimizations involving that trace.

A rule of thumb for the population size is to use ten times the number of parameters in the solution [133]. Since we are only trying to optimize a single parameter (cache size), this implies a minimum population of size 10. If our framework selected fewer points, we iteratively selected points in the center of the largest gap in the curve until we reached 10. It is also possible to optimize the hyper-parameters of the techniques in our framework to select a desired number of points, but we did not explore hyper-parameter optimization in this work.

5.6.2 Acceleration Rate

We evaluated our experiments using the overall *acceleration rate* ($\overline{AR}$) [119] to quantify the increased convergence speed when using our framework to select an initial population for evolutionary algorithms. This metric compares the number

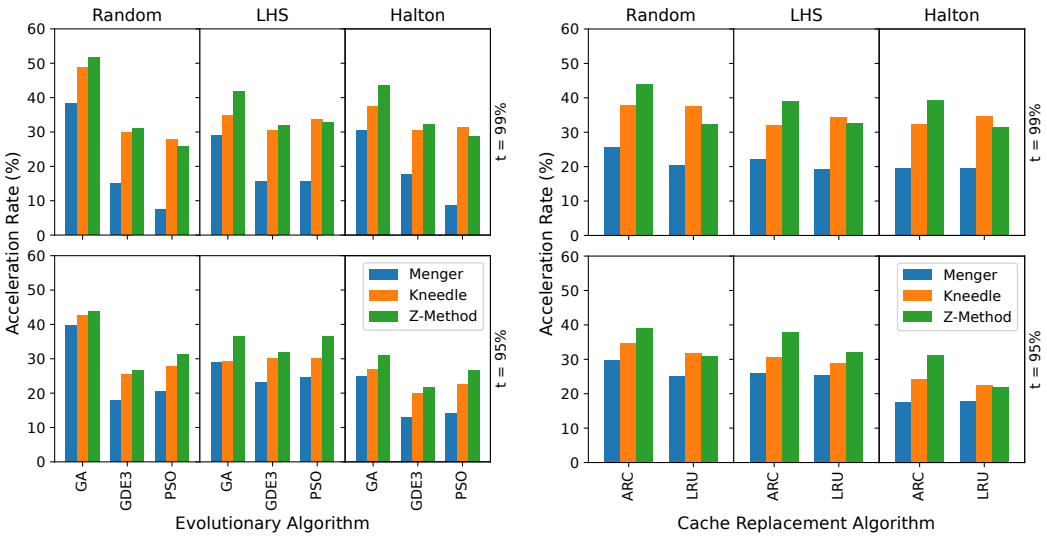
of function calls (NFCs) made by two separate sets of optimization problems. For our purpose, the NFCs will correspond to the number of epochs (iterations) an evolutionary algorithm takes to converge. Each optimization problem uses an evolutionary algorithm to optimize the IOPS/$ of a cache and has several parameters: an input trace, an evolutionary algorithm, a knee-detection algorithm, a population-initialization technique, a threshold for the value-based stopping criterion, and a cache-replacement algorithm. The $\overline{AR}$ compares two sets of problems and reports the percent difference in convergence speed. For all of our evaluations, we compared a set of problems PM that use our multi-knee detection framework for population initialization, against a set PO that uses some other technique for initialization. The $\overline{AR}$ is calculated as follows:

$$\overline{AR} = \left(1 - \frac{\sum_{i=1}^{|PM|} NFC(PM_i)}{\sum_{i=1}^{|PO|} NFC(PO_i)}\right) \times 100\% \tag{5.4}$$

i.e., an x% $\overline{AR}$ implies an x% reduction in running time.

We evaluated 151 traces, four population initialization techniques, two cache-replacement algorithms, and three variations of all other parameters, resulting in 32,616 total problems and over three million cache simulations. The overall $\overline{AR}$ achieved using our multi-knee detection framework for population initialization was 34%. In the remainder of this section, we show the effects of each configuration parameter by creating subsets of the total problems via parameter constraints.

Figure 5.10 presents bar plots showing the $\overline{AR}$ achieved using our multi-knee detection framework for population initialization. Each bar represents a comparison between two sets of optimization problems, PM and PO, which are different sets for each bar, depending on the constraints of the axes and legend.



(a) Acceleration rates using GA, GDE3, and PSO algorithms.

(b) Acceleration rates using ARC and LRU cache replacement.

Figure 5.10: The acceleration rate ($\overline{AR}$) achieved using our multi-knee detection framework for population initialization vs. other techniques. The height of each bar represents the acceleration rate (y-axis), where higher is better. The bottom row represents experiments using the objective threshold $t = 95\%$; the top row is those using $t = 99\%$ (the most challenging threshold to meet). Each group of bars, from left to right, shows the $\overline{AR}$ using our framework for population initialization with the baseline (Menger, in blue), with Kneedle (orange), and with our Z-Method (green), each evaluated against three different initialization methods on the x2-axis (Random, LHS, and Halton, at the top of the figure). (a) shows the $\overline{AR}$ for three different Evolutionary Algorithms on the x-axis (GA, GDE3, and PSO), with results included from both LRU and ARC cache replacement algorithms. (b) shows the same results as in (a) but separated by LRU vs. ARC, with results included from all three Evolutionary Algorithms.

PM uses our multi-knee detection framework for population initialization with a knee-detection algorithm identified by the color of the bar (Menger, Kneedle, or Z-Method). PO uses the initialization technique shown on the x2-axes (top of figure): a pseudo-random number generator (Random), Latin Hypercube sampling (LHS), or Halton sequences. The value of the objective threshold t stopping criterion for both problem sets is shown on the y2-axes.

In each plot, we show three knee-detection algorithms: Menger, Kneedle, and Z-Method. The bars labeled "Menger" represent a baseline knee-detection method; it was the least accurate of those depicted. "Kneedle" is the robust version that we optimized for this scenario; we employed Global RDP (gRDP) before using Menger or Kneedle, reducing noise in the MRCs and improving knee detection. After the knee-detection phase, we applied the post-processing filters described in Section 5.3.3 to refine the selected knees.

We varied the objective-threshold stopping criterion *t* between 99%, 98% (omitted in Figure 5.10 for brevity), and 95% for all configurations. We observed that a lower threshold usually yielded a lower $\overline{AR}$. A higher threshold makes optimization more difficult by requiring a solution closer to the most optimal value. Thus, a lower threshold increases the number of acceptable solutions, giving less-informed population initialization techniques a better chance at finding a solution.

Figure 5.10a shows the effects of using different evolutionary algorithms. The optimization problems analyzed in each group of bars are shown on the x-axis: Genetic Algorithm (GA), Generalized Differential Evolution 3 (GDE3), and Particle Swarm Optimization (PSO). Results in this figure include problem sets with configurations using both ARC and LRU cache replacement. Therefore, in the upper leftmost subfigure of Figure 5.10a, the first blue bar on the left represents, for the 151 traces, the $\overline{AR}$ for all optimization problems using: (1) our framework with Menger for population initialization vs. Random, (2) using a Genetic Algorithm (GA), (3) with either ARC or LRU cache replacement, and (4) an objective threshold $t = 99\%$.

Z-Method is our novel method designed for this task; it outperformed Kneedle in most cases. Kneedle was still competitive, usually only a few percent behind Z-Method and even winning by a small margin in three out of the 18 configurations displayed. (It should be noted that Kneedle benefited greatly from our framework's pre- and post-processing filters, and that Kneedle on its own yielded much poorer results; see Section 5.2.2). We also saw a wide range of $\overline{AR}$ values in these configurations, from under 10% $\overline{AR}$ (Menger with PSO) to over 50% $\overline{AR}$ (Z-Method with GA). Lastly, ranking the $\overline{AR}$ from highest to lowest yields GA (best), GDE3, and then PSO. We attribute this ranking to the difference in effi-

ciency of the evolutionary algorithms. The more efficient algorithms have a lower $\overline{AR}$ because they are less sensitive to the initial population.

Figure 5.10b shows the effects of using either ARC or LRU cache replacement. Results in this figure include problem sets with configurations using any of the three evolutionary algorithms. The most relevant difference between ARC and LRU is that LRU MRCs always decrease monotonically, while ARC can have both convex and concave regions (*i.e.*, can go up and down). This poses a unique challenge for knee detection, since a curve can have a knee that is less optimal than a previous point. Said differently, in ARC and similar non-stack cache algorithms, counterintuitively, adding more cache can actually *hurt* performance. This non-monotonic property can also distort many metrics that knee-detection algorithms use to detect or rank points. Z-Method significantly outperformed Kneedle for all ARC cases in Figure 5.10b, while Kneedle beat Z-Method somewhat for all but one of the LRU configurations. This is to be expected, as Z-Method was originally designed with cache optimization for non-monotonic MRCs in mind, enforcing constraints in a grid-like pattern to prevent proximal knees. Conversely, Kneedle was designed under the assumption of monotonicity. Again, we note that our pre- and post-processing filters were essential to Kneedle's good results.

The population-initialization techniques did not alter any of the previous trends we observed in Figure 5.10, but we did see an approximately 5% difference in $\overline{AR}$ between the worst and best case: Random performed the worst, followed by Halton, and then LHS. These results are consistent with those seen in previous studies [110, 145, 12].

5.7 Chapter Conclusion

The many configurations of multi-tier caching systems produce a wide range of performance and costs. As the configuration space continues to grow due to advancements in caching and storage technology, exploring the space through physical experiments or traditional simulation becomes infeasible.

We introduced the novel concept of applying multi-knee detection to MRCs using a framework for selecting key points, reducing the cost of exploration significantly. We presented Z-Method, an algorithm that robustly and efficiently identifies multiple key points in MRCs with minimal overhead. We also designed a recursive algorithm that enables any single-knee-detection algorithm to find multiple knees. We demonstrated that our framework using Z-Method can be applied to reduce the total number of points required to identify optimal two-tier cache

configurations by an average factor of approximately $5.5\times$ for ARC and $7.7\times$ for LRU compared to naïve approaches. Finally, we evaluated our framework across a highly diverse set of configurations and datasets for the additional application of seeding the initial population of evolutionary algorithms, showing an overall acceleration rate of 34% compared to commonly used population-initialization techniques.

Chapter 6

Pontoon: Single-Pass Live VM Migration via Shared CXL Memory

Live virtual machine (VM) migration is a fundamental capability in modern data centers, supporting maintenance, load balancing, and fault recovery without disrupting running services. However, as VM sizes grow faster than network bandwidth, traditional migration techniques face increasing scalability challenges. Pre-copy migration requires iterative retransmission of dirty pages, inflating migration time and network load. Post-copy migration avoids retransmission but exposes VMs to demand-paging latency and failure risk.

We present Pontoon, a rack-local fast path for live migration that uses shared CXL memory and reframes migration as a memory-placement problem. Pontoon implements transparent guest-memory tiering and consists of three phases: single-pass demotion, stop-and-flush, and background promotion. The source host demotes each DRAM-resident page to shared CXL at most once, remapping it so subsequent writes go directly to the shared device. During blackout, Pontoon uses an efficient writeback instruction to flush cached VM data to CXL, making VM memory visible to the destination and avoiding pre-copy's expensive final guest-memory transfer. After resumption, the destination promotes memory back to local DRAM in the background, using migration telemetry to improve post-migration performance by prioritizing recently active pages for promotion.

We implemented Pontoon in QEMU/KVM and evaluated it on AMD EPYC servers connected by a CXL 2.0 shared-memory pool. On 64 GB VMs, Pontoon improved base migration time by 2.1–9.4× over QEMU's TCP and RDMA migration paths and reduced blackout by roughly an order of magnitude. More importantly, Pontoon's performance remained stable far beyond dirty-page rates

where network-based pre-copy failed to converge.

6.1 Introduction

Live virtual machine migration is a fundamental cloud capability [122, 106, 85, 57, 155]. VM migration enables regular maintenance, security updates, and load balancing without disrupting running services. Google has reported that by 2018, it was performing over 1 million live migrations monthly, with software and firmware updates as the primary driver [122]. As hardware, firmware, and security update cycles accelerate, migration frequency continues to grow.

Live-migration approaches struggle to keep pace. Cloud VM sizes have grown roughly 7× over the past decade, from predominantly <8 GB VMs to 32–64 GB and larger, while network bandwidth has increased only 4× (§6.2.1). The largest VM instances now reach 32 TB of memory [107, 6]. Migrating a modern VM takes proportionally longer and consumes more network bandwidth, a growing operational challenge.

Live migration imposes two forms of disruption: *blackout*, where the VM is completely paused, and *brownout*, where its performance is degraded by contention for memory and network bandwidth. Even sub-second blackouts can trigger distributed service failures, while brownout may violate a Service Level Agreement [38, 13, 117]. Consider a 64 GB VM migrating over a 100 Gbps link: even with RDMA, blackout lasts 276–281 ms and the migration-induced brownout window lasts 10.1–46.2 seconds; active workloads can further extend both blackout and brownout (§6.4).

Existing live migration techniques face significant scalability challenges for large VMs, especially under write-intensive workloads. The most widely used approach is *pre-copy*, in which the source machine iteratively copies the guest VM's memory to the destination over a network while the guest continues to run [114, 33]. Because the guest remains active, pages modified after being copied must be retransmitted later, which for large VMs or workloads with high page-dirtying rates can substantially extend migration time and increase brownout. Once the remaining dirty set falls below a low migration threshold, the VM is paused for a final *stop-and-copy* phase when the remaining dirty pages and CPU/device state are transferred; the VM is *blacked out* and unusable. Although the threshold bounds the amount of memory copied during blackout, it is still costly for latency-sensitive services.

An alternative is *post-copy* migration, which immediately pauses the guest to

transfer only CPU and device state, and then pages in memory on demand using remote faults [64]. Post-copy reduces blackout by eliminating the guest-memory transfer that dominates pre-copy's stop-and-copy phase, but shifts this cost to the destination. Remote page faults are inherently in the guest's critical path, degrading performance until memory is fully fetched from the source. Post-copy is also less robust: once execution resumes on the destination, the VM's latest state is split across two machines, so a server or network failure can leave the VM unrecoverable unless there are separate recovery mechanisms. Thus, production systems of major cloud providers rely primarily on pre-copy [122, 63].

CXL (Compute Express Link) is an emerging interconnect standard for cache-coherent memory sharing [36] that is now entering deployment [70]. Cloud providers are actively exploring the path from single-host memory expansion toward multi-host memory pooling and sharing [16, 175, 168, 167]. The availability of a shared CXL memory pool can fundamentally change live VM migration. When source and destination hosts have access to a common pool, our key insight is that migration becomes a memory-placement problem. We introduce Pontoon, a system that treats the shared CXL pool as a "landing pad": it *demotes* pages from source DRAM to shared CXL in a single pass, transfers ownership, and then *promotes* hot pages to destination DRAM lazily (Figure 6.1). This approach eliminates retransmission entirely: once a page lands in CXL, subsequent guest writes are directed to the shared mapping rather than recopied over the network.

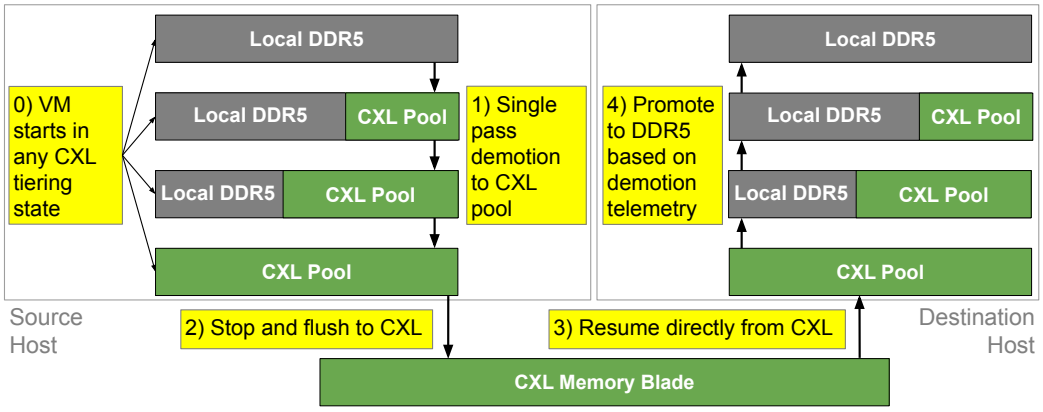


Figure 6.1: Pontoon overview: VMs can start in any state of CXL tiering, and Pontoon progressively demotes pages to CXL pool memory in a single pass. Once a VM is entirely on CXL, Pontoon flushes its state and immediately resumes on the target host, where it progressively resumes tiering.

Pontoon operates on VMs in any state of CXL tiering. A VM with all memory in local DRAM undergoes a single-pass demotion to CXL; one already tiered into CXL requires proportionally less work. In the limit, a fully CXL-resident VM migrates with only a cache flush and state transfer at near-zero cost. After resuming, the destination progressively promotes hot pages back to local DRAM, seeded by demotion telemetry collected during migration, including dirty-page tracking from the source. This allows Pontoon to prioritize pages that were recently modified during migration rather than rediscovering hotness from scratch after the VM resumes. Because CXL memory pools are rack-local resources rather than fabrics that span an entire data center, Pontoon is best viewed as a rack-local fast path within a broader migration framework rather than a universal replacement for network-based migration. When source and destination share a CXL pool, Pontoon can substantially reduce migration time and blackout; when they do not, TCP or RDMA migration remains as a usable, albeit lower-performance, alternative.

A key enabler is the `WBNOINVD` (Write-Back, No Invalidate) instruction, now available on modern CPUs [172, 164]. Current CXL 2.0 hardware lacks automatic inter-host cache coherence, so dirty cache lines must be explicitly flushed before the destination accesses shared memory. Unlike traditional cache flush instructions that evict data, `WBNOINVD` writes dirty lines to memory while preserving them in cache, maintaining locality for any immediate accesses on the source. This enables a *stop-and-flush* phase that replaces pre-copy's stop-and-copy: once guest memory has been moved into shared CXL, the final pause requires flushing dirty cache lines and transferring CPU/device state, but no additional guest-memory copying.

We evaluated Pontoon on 64 GB VMs using synthetic memory-churn workloads and nine CloudSuite workloads. With an idle guest, Pontoon completed migration in 4.9 seconds, 2.1–9.4$\times$ faster than QEMU's TCP, RDMA, and RDMA-pinall migration paths, while reducing blackout from 267–281 ms to 29 ms. Under load, Pontoon behaved as a nearly fixed-cost migration mechanism: it transferred each page at most once, kept blackout nearly constant, and continued to complete migration under dirty-page rates where network-based pre-copy failed to converge. Our evaluation also showed that Pontoon's placement policies matter beyond raw migration time: telemetry-guided promotion improves post-migration recovery, while zero-page-first demotion reduces migration-window brownout by keeping active memory in local DRAM longer. Together, these results show that shared CXL memory can provide a fast, predictable migration path while giving the hypervisor explicit control over blackout, brownout, and recovery.

Pontoon makes three contributions:

(1) Single-pass demotion. The source host copies each DRAM-resident page exactly once into shared CXL memory, remapping it so that subsequent writes go directly to CXL. This eliminates the iterative retransmission of pre-copy.
(2) Stop-and-flush. We leverage `WBNOINVD` to flush dirty cache lines without eviction, then transfer CPU and device state. This avoids post-copy's expensive remote paging after switchover while keeping the memory-transfer portion of blackout bounded by LLC flush time.
(3) Telemetry-guided promotion. We repurpose demotion telemetry, including migration dirty-page tracking, to seed the tiering policy, ensuring that hot pages are the first ones promoted to the target host's DRAM after resumption.

We implemented Pontoon primarily in QEMU/KVM, with a minimal kernel module for fast cache writeback, on commercially available CXL 2.0 hardware. All of our code will be open source and available on GitHub.

Scope. We target diskless VMs, which represent a dominant pattern in cloud deployments: major providers offer instance families that rely solely on network-attached storage [5, 108, 53]. We also assume that hypervisor-bypass networking (*e.g.*, SR-IOV) and its dirty-page tracking are handled separately, as recent work addresses this orthogonal challenge [155]. Finally, Pontoon is best viewed as a rack-local fast path within a broader migration framework rather than a universal replacement for network-based migration.

6.2 Background and Motivation

6.2.1 The Live-Migration Challenge

Cloud VM memory sizes have grown substantially over the past decade, driven by data-intensive applications. Table 6.1 illustrates this shift. In 2017, over 80% of Azure VMs were smaller than 8 GB [35], and Google's Borg traces show a similar concentration of sub-8 GB allocations [138]. By 2019, Azure reported growth in the 8–32 GB tier, though small VMs still dominated [17]. We analyzed a two-week production trace from Microsoft Azure collected in November 2025, where only 6% of VMs were smaller than 8 GB and 56% were 32 GB or larger. Large VMs were common: 20% were 64 GB and 12% exceeded 64 GB.

Memory	Azure '17	Google '19	Azure '19	Azure '25
<8 GB	83%	>90%	61%	6%
8–32 GB	15%	9%	29%	43%
32–64 GB	–	–	–	20%
64 GB	2%	<1%	6%	20%
>64 GB	0%	0%	4%	12%

Table 6.1: Evolution of VM memory distributions.

Network bandwidth on general-purpose VM servers has not kept pace with this growth. NIC bandwidth increased from 50 Gbps to 200 Gbps (4×), while VM sizes grew by roughly 7× on average and much more at the high end. This mismatch makes memory transfer the central scaling bottleneck for network-based migration.

6.2.2 Live VM Migration Techniques

Network-based live migration primarily uses two techniques: pre-copy, which transfers memory before pausing the VM, and post-copy, which defers transfer until after resumption. Both reduce one form of disruption by increasing another.

Pre-copy migration. Pre-copy remains the standard for live migration [114, 33] and is the QEMU default. It first copies guest memory to the destination while the VM runs, then iteratively retransmits pages dirtied since the previous round. Once the dirty set is small enough, the VM is paused for a final *stop-and-copy* phase, during which the remaining dirty pages and CPU/device state are transferred before execution resumes on the destination. Although pre-copy tries to minimize this pause, any guest-memory transfer during blackout is costly: even short blackouts can cascade into stalled or failed operations in dependent services [72].

To track which pages are dirty, the hypervisor must monitor guest writes. QEMU sets write protection so the first write to each region causes a write fault and is logged; the hypervisor scans the dirty bitmap and retransmits marked pages. (While some processors have hardware-assisted dirty tracking, such as Intel page modification logging (PML) [73], it is not available on the AMD CPUs we used for evaluation.) For write-intensive workloads, this process can continue indefinitely: if pages are dirtied faster than they can be transferred, the dirty set does not shrink.

When convergence stalls, systems can abort migration or throttle the guest to reduce the dirtying rate. QEMU has an `auto-converge` parameter that progressively throttles guest vCPUs when migration is not making progress. VMware's XvMotion uses a more targeted mechanism, Stun During Page Send (SDPS), which injects microsecond-scale delays in response to guest writes when necessary [101]. Both approaches force convergence by reducing the guest dirtying rate, intentionally degrading guest execution. Thus, pre-copy's cost has three sources: lower guest CPU performance; retransmission during the iterative phase, which competes for memory and network bandwidth; and stop-and-copy, which pauses the VM to copy the remaining dirty pages.

Post-copy migration. Post-copy takes the opposite approach: pause immediately, transfer CPU/device state, resume on the destination, and fetch pages on demand [64]. This transfers each page at most once, but shifts the overhead onto the resumed VM. Every access to an untransferred page stalls, replacing pre-copy's background contention with foreground stalls on the guest's execution path.

Post-copy also has a weaker recovery model unless augmented with fault-tolerance machinery. In standard post-copy, the destination may hold newer VM state after resumption while the source is stale, so losing the destination is a critical VM failure [64]. PostCopyFT addresses this with reverse incremental checkpointing, sending destination-side memory and execution updates back to the source during migration [47]. Although PostCopyFT reports low average migration-time overhead, its fault-tolerance mechanism requires dirty-page tracking, checkpoint capture, reverse state transfer, and buffering outgoing packets until checkpoints commit. Thus, robust post-copy is possible but complicates the protocol and shifts part of the problem into checkpointing and I/O consistency. Google's production fleet uses pre-copy [122]; hybrid approaches that limit pre-copy iterations before switching to post-copy are possible but we are not aware of public documentation of their use in production.

RDMA-based migration. RDMA improves migration efficiency by offloading data transfer to the NIC, reducing CPU overhead and achieving higher bandwidth than TCP [68]. However, RDMA does not change the fundamental migration algorithm: pre-copy still retransmits dirty pages, and post-copy still incurs remote page faults. RDMA accelerates the data plane but cannot eliminate the control-plane bottlenecks inherent in network-based migration.

6.2.3 Compute Express Link (CXL)

Compute Express Link (CXL) is an open industry-standard interconnect that provides cache-coherent memory access between processors, memory devices, and accelerators [36]. CXL enables memory pooling and expansion, reducing data center costs by improving memory utilization [86, 100, 174, 153]. Cloud providers are explicitly considering CXL pooling for VMs: Azure now offers VMs with CXL-attached memory in private preview [70], and Alibaba has deployed CXL memory for databases and LLM inference [168, 167]. Scalable, switchless CXL topologies can further reduce deployment costs by eliminating expensive fabric switches [175].

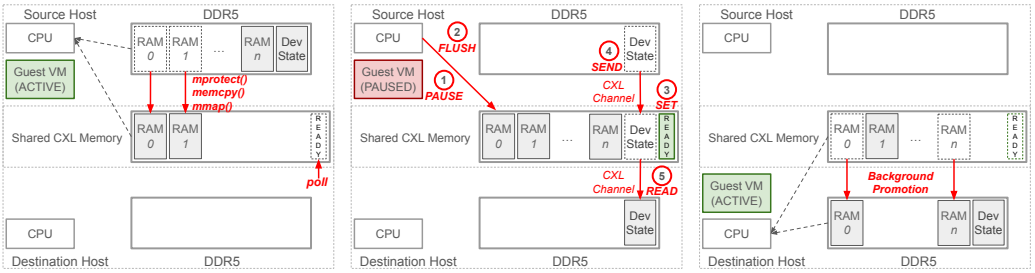
CXL memory exhibits higher access latency than local DRAM, typically 2–3$\times$ depending on the topology and workload [92, 16]. Some latency-insensitive VMs can run entirely on CXL memory [86, 92], while latency-sensitive applications require *memory tiering*: placing hot pages in fast local DRAM and cold pages in slower CXL memory. A substantial body of recent work addresses tiering policies based on migrating pages between tiers [100, 174, 160, 93, 149, 129].

In practice, VMs using CXL exist across a spectrum of memory configurations, from fully DRAM-resident to fully CXL-resident, with tiered configurations in between. Live migration for CXL-enabled data centers must therefore support VMs in any of these states, as illustrated in Figure 6.1.

6.3 Pontoon

6.3.1 Design Overview

Pontoon migrates VMs across a shared CXL memory pool in three phases: *single-pass demotion*, *stop-and-flush*, and *background promotion* (Figure 6.1). A VM may be fully DRAM-resident, CXL-resident, or tiered between the two. Pontoon uses the same algorithm in all cases, but only DRAM pages need demotion, during which the source copies each DRAM-resident page to CXL and remaps it to redirect subsequent guest writes. The stop-and-flush phase pauses the VM, writes dirty cache lines to CXL, and transfers CPU and device state. The destination then resumes directly from CXL and can promote pages to local DRAM in the background, optionally using telemetry collected during demotion to prioritize recently modified pages. This design eliminates retransmission: each DRAM-resident page is copied once, and pages already in shared CXL require no data movement.



(a) Single-pass demotion (b) Stop-and-flush (c) Background promotion

Figure 6.2: Implementation of Pontoon's three migration phases. (a) Single-pass demotion: each DRAM page is copied to CXL, write-protected via `mprotect`, and remapped via `mmap`; subsequent writes go directly to CXL. (b) Stop-and-flush: the VM pauses, `WBNOINVD` flushes dirty cache lines, and CPU/device state transfers over the CXL channel. (c) Background promotion: the destination VM resumes immediately; a background thread optionally promotes hot pages to local DRAM.

6.3.2 Design Constraints

Pontoon's design is shaped by two properties of the target environment: DAX-backed shared CXL memory and explicit cache-coherence management.

DAX-backed shared memory. CXL memory can be exposed either as system RAM or through DAX mappings. In system RAM mode, the host manages the CXL capacity as ordinary memory: it chooses physical pages to back QEMU's allocations, and file-backed access through the normal path is subject to page-cache behavior. That model is unsuitable for Pontoon because migration requires both QEMU processes to coordinate through explicit offsets in the same shared device and to control exactly which guest-memory ranges are backed by CXL. Therefore, Pontoon uses DAX mappings, giving QEMU direct load/store access to CXL without page-cache mediation and allowing the source and destination to map matching device offsets. This choice affects the granularity at which Pontoon can safely protect, copy, and remap memory. Syscalls such as `mprotect()` and `mmap()` must operate on regions that adhere to the DAX region's alignment, introducing a potential mismatch: the alignment for CXL Type 3 memory devices is normally 2 MB, while the hypervisor may use smaller page sizes. Thus, DAX alignment is the minimum granularity of our copy and remap process, *i.e.*, the size of the memory chunks we operate on. In practice, major cloud providers use huge

pages for large VMs, so the required alignment is consistent with how those VMs are provisioned [86]. Pontoon arranges guest-memory ranges on the CXL device so that remapped chunks satisfy DAX alignment.

Inter-host coherence. CXL 2.0 does not provide automatic cache coherence between hosts; CXL 3.0 defines this capability, but compliant hardware was not available for this work.[1] With current hardware, both machines must explicitly coordinate when accessing shared memory. The writer must flush dirty cache lines to CXL before signaling the reader, and the reader must invalidate to avoid consuming stale copies. For guest memory, this means any pages dirtied after their remapping must have their corresponding cache lines written back before the destination accesses guest memory. For synchronization metadata and host communication over CXL, Pontoon similarly issues explicit flushes and invalidations as needed.

6.3.3 QEMU Integration

We extended QEMU/KVM v9.2.92 to enable live VM migration using both our CXL-based and tiered-memory migration techniques over a shared CXL Type 3 device in DAX mode. Our implementation functions entirely in user space, with an optional kernel module for the WBNOINVD cache flush optimization that dramatically reduces blackout time.

A RAMBlock is QEMU's internal structure for a contiguous region of guest RAM, including its host backing memory and metadata for migration and dirtiness tracking. Pontoon implements two CXL-based data-movement mechanisms. RAMBlock contents move between DRAM and shared CXL memory using our page-remapping process during demotion and promotion. Serialized CPU and device states and migration control messages use a separate CXL-based channel that stores serialized bytes in slots sized to the CXL DAX alignment. As the source fills each slot, it is marked ready. The destination polls this metadata and uses an *eventfd* to wake the channel reader when data is available.

An overview of our CXL-based migration implementation in QEMU/KVM is presented in Figure 6.2. Each of its three phases exploits both multi-threading and the shared-memory semantics provided by CXL. Our implementation cleanly

[1] Thus, the performance of hardware-coherent sharing in our scenario is unknown; it is possible that the cost of automated coherence would outweigh its benefits.

integrates with QEMU's existing migration infrastructure while significantly improving overall migration efficiency.

6.3.4 Transparent CXL Tiering

QEMU does not currently support hypervisor-managed memory tiering, so we first implemented that feature. We manage page placement entirely within QEMU, completely transparent to the guest VM, so the guest sees both memory sources as a single device and is unaware of whether pages are backed by local DRAM or remote CXL memory.

Each of the guest's RAMBlocks has a *host* pointer whose virtual address refers to the physical memory that backs the block, allocated upon VM initialization according to its configuration. For main memory, QEMU allocates a contiguous region of anonymous heap. We implemented tiering by remapping the desired amount of that heap-backed region to the CXL memory device via `mmap()`. To track page placement, we added a new bitmap to the RAMBlock struct to indicate which pages reside in CXL.

Our tiering implementation is mostly naïve, extending guest memory capacity with CXL without attempting to place pages based on access patterns or other metrics that might optimize performance. However, we do apply a simple optimization that maps the upper address range of system memory's RAMBlock *host* pointer to CXL memory. Since Linux's memory allocator prefers to consume lower addresses first, this optimization ensures that the guest uses all available DRAM before touching CXL.

6.3.5 Single-Pass Demotion

The demotion phase sends each DRAM-resident guest page to shared CXL memory exactly once, as shown in Figure 6.2a. The key observation is that both hosts can access a page after it is placed in CXL. For each page, the source write-protects the current host mapping with `mprotect()`, copies the page to CXL, and replaces the original mapping with a CXL-backed one using `mmap()` with `MAP_FIXED`. The `mprotect()` call makes the old DRAM read-only during the transition, preventing writes from reaching stale memory while the copy and remap are in progress. After the remap, future writes update the shared CXL-backed mapping directly.

If, during the transition, the guest writes to a protected page, a host-side protection fault occurs and exits through KVM to QEMU. In Pontoon, we handle these

faults as transient remapping conflicts: while demotion is active, QEMU converts the fault from `EFAULT` to `EAGAIN`, causing KVM to retry the vCPU rather than treating it as fatal. This retry path takes on the order of tens of microseconds, comparable to the time required to copy and remap a 2 MB DAX-aligned region. As a result, the copy and remap usually complete before the guest retries the write. Correctness does not depend on this timing: a write-heavy guest may fault more than once, but each retry observes either the protected old mapping or the completed CXL mapping, never a partially copied writable mapping. In the worst case, the vCPU stalls until the protected region has been copied and remapped.

Zero-page-first demotion. We implemented a demotion optimization that processes zero-filled chunks before non-zero DRAM-backed ones. Linux normally allocates physical pages for anonymous memory lazily: a zero-filled virtual range can exist before DRAM is physically allocated, which occurs only when the range is written. That optimization does not apply to source-side CXL demotion, because DAX memory already has fixed physical backing. Thus, Pontoon must still write zero-filled chunks to the DAX device.

The benefit of zero-page-first demotion is ordering: Pontoon delays remapping *non*-zero memory, which is more likely to contain the guest's active working set, so we keep the guest on local DRAM for more of the migration window. Zero-page detection also helps during destination-side promotion. When Pontoon promotes a chunk known to be zero, it can replace the CXL mapping with anonymous memory instead of redundantly copying zeroes from CXL into DRAM. Thus, Pontoon again exploits Linux's lazy allocator.

6.3.6 Stop-and-Flush

Once all DRAM-resident pages have been demoted to CXL memory, the VM enters stop-and-flush. The guest is paused while CPU and device states are transferred to the destination. Because CXL 2.0 lacks automatic inter-host cache coherence, any pages dirtied after their remapping must have the corresponding CPU cache lines flushed before the destination accesses guest memory.

The stop-and-flush phase is complex and requires careful coordination between the hosts to ensure that flushing RAMBlocks, transferring device states, and pausing and resuming VM execution are done in a precise order. Figure 6.2b depicts this phase and highlights the ordering.

The first step pauses the guest VM so it cannot modify device states after they

have been transferred and so RAMBlock memory contents are not modified after the final flush.

The second step involves flushing the CPU cache lines corresponding to any pages that were written after they were remapped to CXL memory in the previous phase. We implement two different paths for flushing:

WBNOINVD fast path. The naïve approach to cache flushing is to iterate over the dirty pages and issue `CLFLUSHOPT` for every cache line that may contain modified guest data. To avoid this per-line flush cost, we invoke a whole-cache writeback instruction through a small kernel module that exposes `/dev/wbinvd`. When available, the module executes `WBNOINVD` on all CPUs via `on_each_cpu()`, ensuring all dirty cache lines are written back to CXL memory while preserving clean copies in the cache. If `WBNOINVD` is unavailable, the module instead executes `WBINVD`, which writes back dirty cache lines and invalidates the caches. This fast path avoids issuing one `CLFLUSHOPT` per cache line in the dirty set, which is the dominant cost of iterative flushing for large dirty sets. Unlike iterative flushing, the whole-cache path uses a constant number of explicit flush instructions, reducing cache-write-back time from tens of milliseconds with `CLFLUSHOPT` to approximately 0.4–7.2 ms with `WBNOINVD` in our experiments (see § 6.4.8). When `WBNOINVD` is used, preserving clean cache lines also maintains locality for any immediate accesses on the source after the flush, such as rollback scenarios.

CLFLUSHOPT fallback. If neither whole-cache writeback instruction is available, we fall back to iterative `CLFLUSHOPT` instructions, going over each RAMBlock's dirty bitmap and individually flushing dirty pages' cache lines.

The third step begins after all RAMBlocks have been flushed. At this point, the source flips the *READY* bit to indicate that all guest RAMBlock data is visible in CXL and the destination can proceed with the final restore path. It is critical to wait for RAMBlocks to be flushed before flipping *READY*, because some devices access guest RAM while restoring their state. For example, virtio-net rebuilds its `vring` structures in guest memory during state restoration. Thus, the device state would be corrupted if the destination began to restore it from the CXL channel before RAM was flushed.

The fourth step sends CPU and device state, along with QEMU control messages, over the CXL-based channel. Because the *READY* bit has already been set, the destination knows that guest memory is visible in CXL before it begins

consuming this serialized state.

The fifth and last step begins when the destination observes the *READY* bit. This signal indicates that all guest RAM has been copied or remapped into CXL and is visible to the destination. The background polling thread notifies the main thread via the *eventfd*, after which QEMU reads the remaining CPU and device state from the CXL channel, completes restore, and resumes the guest on the destination.

6.3.7 Background Promotion

At the destination, guest execution initially resumes entirely from CXL memory. Pontoon can then keep some guest memory in CXL or promote selected pages into local DRAM in the background. Unlike post-copy migration, promotion is not triggered by guest page faults: guest accesses can complete directly from CXL while the destination promotes pages asynchronously according to its promotion policy.

Telemetry-guided promotion. Pontoon guides promotion with telemetry collected during the demotion phase. Pages marked dirty during demotion were recently written and are likely to be accessed again soon; promoting these first may improve post-migration performance. This repurposes QEMU's existing dirty-tracking infrastructure: rather than driving retransmission as in pre-copy, dirty bits inform the destination's promotion policy.

We provide simple promotion policies that use the dirty bitmaps: *none*, *all*, *dirty*, and *dirtyfirst*. The *none* and *all* policies promote no or all pages from CXL to DRAM, respectively. The *dirty* policy solely promotes pages that were written to during the migration process, as indicated by each RAMBlock's dirty bitmap. The *dirtyfirst* policy first promotes the dirtied pages, and then all of the remaining pages.

Promotion reuses the demotion page-remapping process in reverse, moving data from CXL back to local DRAM. As such, we use the same `mprotect()`, copy, and `mmap()` process, similar page fault handling, and multi-threading.

6.4 Evaluation

We compared Pontoon to TCP- and RDMA-based migration using a synthetic memory-churn benchmark and CloudSuite benchmarks on 64 GB VMs. Our key

findings are:

- **Pontoon avoids pre-copy convergence limits:** Pontoon transfers each guest page at most once, so migration time, blackout, and data movement stayed stable as guest dirtying rates increased. TCP and RDMA instead retransmitted dirty pages and failed to converge under high dirty-page pressure (§§ 6.4.3 and 6.4.4).
- **TCP and RDMA fail for workload-dependent reasons:** TCP is limited mainly by effective migration bandwidth, while RDMA is also sensitive to dirty-page locality and dynamic memory-registration cost. CloudSuite exposed both effects: Web Search exceeded TCP's convergence limit, while Data Caching fragmented RDMA migration into small writes and kept RDMA from converging (§§ 6.4.3 and 6.4.4).
- **Pontoon's migration telemetry improves CXL recovery:** Using the dirty bitmap to promote pages dirtied during migration first reduced post-migration recovery time by 46.2% and lost work by 90.3% in our worst-case placement experiment (§ 6.4.5).
- **Pontoon's CXL demotion order reduces brownout:** Our zero-page-first demotion kept the working set in local DRAM longer, reducing migration duration by 55.7% and migration lost work by 74.5% in our worst-case placement experiment (§ 6.4.6).
- **CXL shifts the bottleneck from convergence to contention:** Under maximum-stress benchmarks, Pontoon still completed migration when network-based pre-copy failed to converge. However, high thread counts and larger VMs stretched demotion and promotion by forcing the guest and migration threads to share CXL bandwidth, producing extended brownout rather than convergence failure (§ 6.4.7).

6.4.1 Experimental Setup

We used a three-server testbed consisting of a workload driver, a migration source, and a migration destination. The driver was a single-socket Intel Xeon Platinum 8573C CPU with 52 cores (104 hardware threads). The source had a single-socket AMD EPYC 9825 Turin CPU with 144 cores (288 threads). The destination used a single-socket AMD Turin engineering-sample CPU with 128 cores (256 threads).

The source and destination connected to a shared CXL pool over an $\times 8$ PCIe 5.0 CXL 2.0 link. The CXL device exposed its memory to both servers in DAX mode. Each host had two 100 Gbps CX6 NICs connected to a 100 Gbps switch. All hosts ran Ubuntu 24.04, and the source and destination ran virtual machines using QEMU/KVM v9.2.92. Because the source and destination used different CPUs, we configured QEMU to expose the `EPYC-Milan-v2` guest CPU model, which is the newest one fully compatible with both machines.

Guest VMs were configured with 8 vCPUs, running Ubuntu 24.04. Unless otherwise noted, experiments used 64 GB VMs to represent typical cloud workloads. We pre-faulted all VM memory so that freshly spawned experimental VMs more closely resembled longer-running cloud VMs, which typically have fewer untouched zero pages.

We compared Pontoon against the migration methods currently supported by QEMU: (1) TCP-based, using QEMU's default socket transport, (2) RDMA-based, using QEMU's default dynamic-registration path, and (3) RDMA-based, with all guest memory pre-pinned using QEMU's *pin-all* setting. For all transports, we used QEMU's default migration parameters unless explicitly stated otherwise.

6.4.2 Metrics

We measured three metrics: (1) *total migration time* from initiation to destination VM resumption, (2) *blackout duration*, the interval during which the source VM is paused, and (3) *data transferred* in bytes. Host-side metrics were collected via QEMU's migration monitoring interface.

Table 6.2 presents baseline migration metrics for all transports with an idle guest. The measured data transferred was 67.1 GB for each transport: the VM's migratable RAM/ROM state plus serialized CPU/device state and migration protocol data, with no retransmitted dirty guest pages.

Transport	**Migration Time (ms)**	**Blackout (ms)**
CXL	4925.8 ± 135.0	29.0 ± 2.5
TCP	22423.2 ± 2521.5	267.4 ± 40.3
RDMA	46173.8 ± 1597.5	280.6 ± 28.6
RDMA-pinall	10105.8 ± 37.4	276.4 ± 3.4

Table 6.2: Baseline idle-guest migration metrics for 64 GB VMs, averaged over five runs.

6.4.3 Synthetic Memory-Churn Benchmark

We used a synthetic memory-churn benchmark, running inside the guest VM, to isolate how each migration transport behaved under controlled guest-application dirty-page rates. The benchmark allocates a 60 GB working set within the 64 GB VM, leaving headroom for the OS and benchmark process. It first-touches every page to make the full working set resident, then launches worker threads that repeatedly overwrite fixed-size chunks of memory. Each write changes the chunk contents, ensuring that repeated accesses continue to dirty pages throughout migration. We controlled the benchmark's maximum aggregate write bandwidth, letting us evaluate each transport at increasing dirtying rates.

We benchmarked two access patterns and two application write sizes. Sequential access dirties contiguous memory, exposing how each transport exploits spatial locality. Uniform random access spreads writes across the working set, stressing sparse dirty-page tracking and retransmission. For each access pattern, we tested both 4 KB and 1 MB writes to evaluate how write granularity affected each transport. Together, these configurations exposed the main differences between the transports: TCP's lower tolerance for application dirtying rates, RDMA's additional dependence on contiguity and batching, and Pontoon's independence from retransmission.

Figure 6.3 shows how each migration transport scaled with application bandwidth across migration time, blackout time, and total data sent. For TCP and RDMA, vertical arrows indicate the application bandwidth where migration failed to converge. These arrows are not shown for CXL, as it is guaranteed to finish migration in a single pass without needing to retransmit any guest memory.

CXL remained stable across all three metrics. Migration time remained near 5 seconds at lower dirty-page rates, then increased to roughly 7–8 seconds at 8 GB/s and above. This increase was caused by bandwidth contention on the CXL de-

vice, since both the migration process and the guest workload access CXL-backed memory during migration. Blackout time remained nearly constant and substantially lower than TCP and RDMA because, unlike TCP and RDMA, CXL does not copy dirty memory during blackout. Finally, CXL sent a fixed amount of data every run because Pontoon uses a single-pass design. If some guest memory were already tiered in CXL before migration, this transfer would decrease in proportion to the fraction already resident in CXL.

For both TCP and RDMA, raising the dirtying rate increased migration time and data sent, until migration eventually failed to converge. However, TCP's lower migration bandwidth resulted in longer migration times and more data sent, and caused TCP to fail earlier. QEMU streams TCP migration data through sockets, keeping the host CPU and kernel networking stack on the data path. The same migration buffers and socket writes are used regardless of chunk size or access pattern, causing TCP to remain stable across benchmarks.

RDMA exhibited more varied behavior. First, total migration time and data sent were reduced for RDMA-pinall. This is expected because *pin-all* avoids the cost of dynamically registering RDMA regions. Second, RDMA was highly sensitive to locality when accesses were small. QEMU's RDMA path is optimized for data staged into 1 MB chunks. During each pre-copy iteration, it walks the dirty bitmap and adds adjacent dirty pages to the staged chunk. If the pages are not adjacent, it flushes the chunk and issues an RDMA write for the data staged so far. Sparse 4 KB uniform writes thus fragmented the migration stream into small writes, reducing effective bandwidth and causing both RDMA configurations to fail to converge at much lower dirty-page rates.

RDMA-pinall with the 4 KB uniform workload also exposed a weakness in QEMU's convergence-threshold heuristic that causes it to overestimate when the VM is ready to pause. This heuristic chooses the threshold partly based on observed throughput, so the first pre-copy round's 10.5 GB/s bulk copy led it to allow roughly 3.15 GB of dirty memory before pausing. Later rounds fell to about 1.2 GB/s once dirty pages became sparse, causing QEMU to pause with too much dirty memory remaining and increasing blackout to multiple seconds. Tuning `avail-switchover-bandwidth` replaces the observed throughput in this calculation with a configured bandwidth value, so setting it to 1 GB/s returned blackout to normal levels. However, in practice this approach would require detailed knowledge of the guest workload pattern and its effective migration throughput.

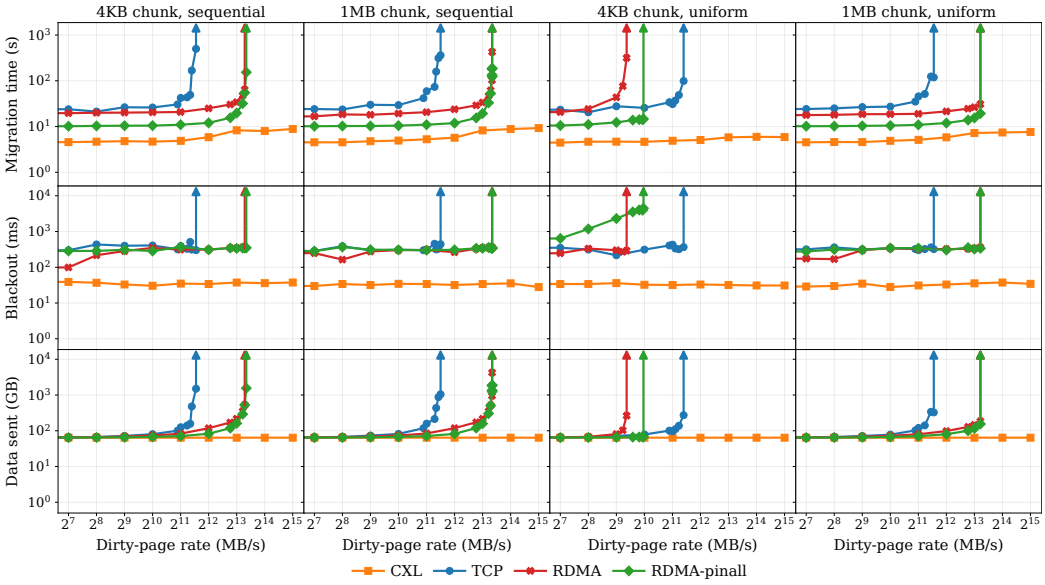


Figure 6.3: Synthetic memory-churn benchmark showing migration time, blackout, and data sent as guest dirty-page rate increases across access patterns and write sizes. Vertical arrows mark the dirty-page rate where transports fail to converge.

6.4.4 CloudSuite

We used CloudSuite to compare Pontoon with TCP and RDMA migration, and to study how realistic workload behavior affected migration performance [46]. The benchmarks represent diverse workloads: Data Analytics (Spark), Data Caching (Memcached), Data Serving (Cassandra), Graph Analytics, In-Memory Analytics, Media Streaming, Web Search (Solr), and Web Serving (Nginx/PHP). Because the Web Serving benchmark has separate database and web components, we report them separately as WebServe (DB) and WebServe (Web), yielding nine total workloads that span compute-intensive, memory-intensive, and I/O-intensive applications with varying memory access patterns and dirtying behavior.

We group workloads into three categories. *Low dirty-rate* workloads (Data Analytics, Media Streaming) dirty memory infrequently, letting traditional migration converge quickly. Garbage collection in *high dirty-rate* workloads (Web Search, Graph Analytics) continuously dirties pages, delaying pre-copy convergence. The remainder fall between these cases or expose transport-specific behavior.

Figure 6.4 compares migration time, blackout, and total data sent for each transport on the nine benchmarks. Across these metrics, Pontoon offered nearly fixed-cost migration. Its migration time remained low, blackout was nearly constant, and data movement was fixed at the VM memory size because each page was transferred at most once. In contrast, TCP and RDMA varied substantially by workload, with migration time and data movement increasing when workloads induced more retransmission; some workload/transport combinations failed to converge (marked by an "*X*").

The Data Caching benchmark exposed a limitation of RDMA migration. This workload is driven by Memcached serving key-value requests, which tend to touch mostly small, scattered objects rather than long contiguous memory ranges. The resulting sparse dirty set matched the behavior of the 4 KB uniform-access synthetic workload in Figure 6.3. As discussed in § 6.4.3, this pattern prevented QEMU's RDMA path from forming its preferred 1 MB writes and instead triggered many small transfers. Our instrumentation confirmed this effect in Data Caching, with average RDMA writes of roughly 8 KB. This reduced RDMA's effective migration throughput below TCP's for this workload, so the dirty-page rate landed below TCP's convergence limit but above RDMA's, causing RDMA to fail to converge while TCP completed.

Web Search showed the opposite case, where the access pattern was favorable to RDMA but the dirty-page rate was too high for TCP. This workload is driven

by Solr query processing over a large in-memory index, producing Java heap allocation and garbage-collection activity. This activity dirties memory in spatially clustered heap regions, allowing QEMU's RDMA path to optimally batch writes. That matched the higher-bandwidth synthetic cases in Figure 6.3. Thus, Web Search failed to converge for TCP because its dirty-page rate exceeded TCP's effective throughput, but completed for RDMA because its access pattern let RDMA achieve a higher bulk-transfer bandwidth.

RDMA without pin-all was sensitive not only to the amount of data transferred, but also to the cost of dynamically registering RDMA memory. In the Web Serving and Data Serving workloads, migration transferred roughly the same amount of data as Media Streaming, and the RDMA writes remained similarly sized, but our instrumentation showed that the dynamic-registration path behaved differently. In Media Streaming, dynamic registration cost about 8% of the migration time, with average registration latency around 23 μs. In Web Serving DB, registration accounted for over 90% of the time, with latency roughly two orders of magnitude higher. This isolated the collapse to the registration path rather than the amount or granularity of data sent. The likely explanation is that Web Serving and Data Serving exposed a more expensive on-demand pinning and RNIC mapping path than Media Streaming, whose file-serving behavior left registration relatively cheap.

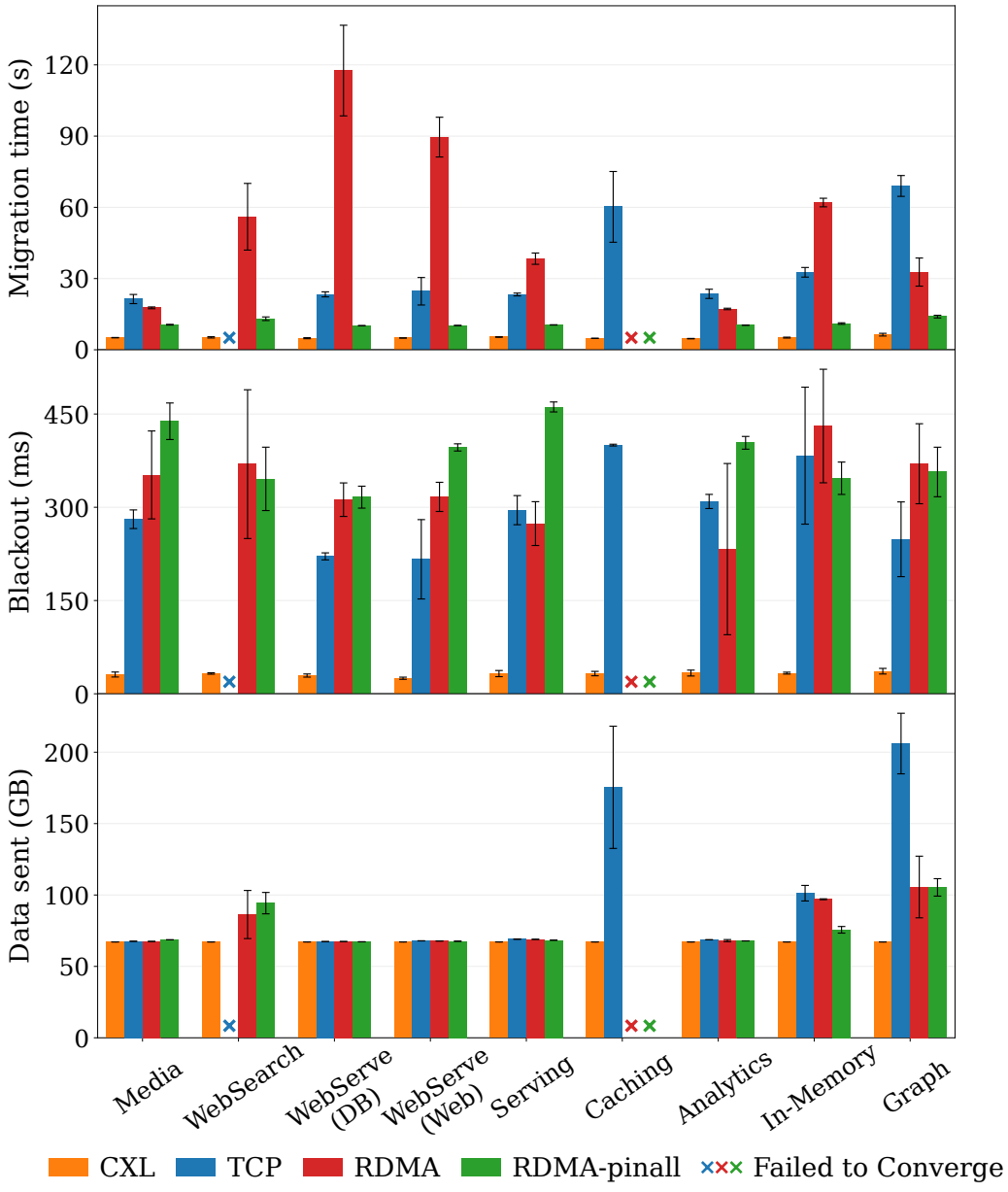


Figure 6.4: Migration performance on nine diverse CloudSuite benchmarks. Pontoon keeps all metrics nearly fixed, while TCP and RDMA vary by workload. "*X*" markers indicate transport/workload pairs that failed to converge.

6.4.5 Telemetry-guided promotion

Figure 6.5 illustrates Pontoon's telemetry-guided promotion policy after the destination resumes execution. The top plot, labeled UNSORTED, shows the baseline policy, which promotes memory in RAMBlock address order. The bottom plot, labeled DIRTY FIRST, shows our telemetry-guided policy, which uses the dirty bitmap collected during migration to promote pages written during migration before promoting the remaining pages.

We created a worst case for address-ordered promotion using the synthetic memory-churn workload with a small 2 GB working set accessed at a very high rate, around 135 GB/s. We placed that working set near the upper end of guest memory, so UNSORTED reached it only near the end of its promotion scan. DIRTY FIRST instead observed that this region was dirtied during migration and promoted it first.

We quantified recovery using two metrics. *Recovery time* is the interval between migration completion and the first throughput sample that reaches the post-migration stable level. *Lost work* is the area between the stable-throughput baseline and the measured throughput curve over the recovery interval, capturing throughput lost while pages are promoted into local DRAM.

Both policies reached roughly 135 GB/s, but their recoveries differed substantially. UNSORTED took 12.8 s to recover and lost 1012.0 GB of work relative to stable throughput. DIRTY FIRST reached stable throughput in 6.9 s, reducing recovery time by 46.2% and lost work by 90.3% to 98.5 GB.

These results reflect a highly favorable case rather than a typical benefit. The advantage of DIRTY FIRST depends on how well the dirty bitmap predicts the post-migration working set, whether those pages would otherwise be promoted late, how sensitive the workload is to CXL performance, and how quickly promotion can move data into local DRAM.

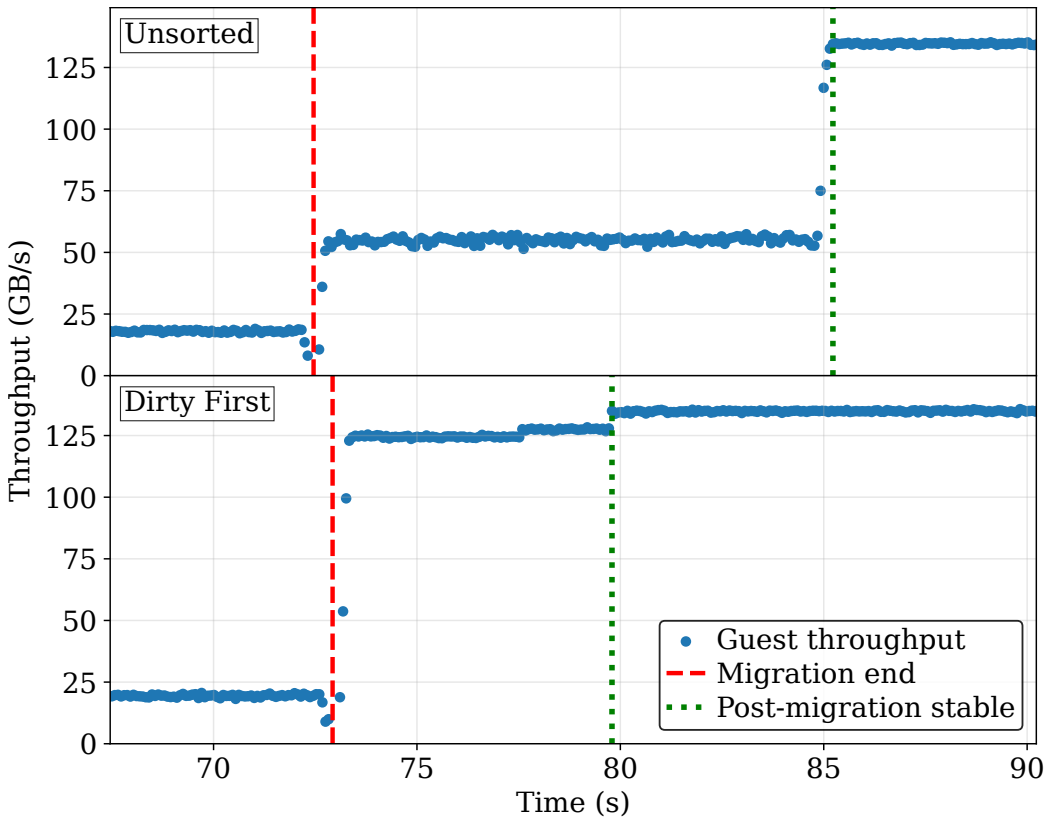


Figure 6.5: Telemetry-guided promotion on a high-throughput 2 GB memory-churn working set placed late in address order, making unsorted promotion unfavorable. Dirty-first promotion reduces recovery time by 46.2% and lost work by 90.3%.

6.4.6 Zero-page-first demotion

Figure 6.6 shows the effect of Pontoon's zero-page-first demotion order during migration. UNSORTED demotes memory in RAMBlock address order, while ZERO FIRST demotes zero pages before non-zero pages. This order keeps the working set in local DRAM longer, reducing CXL bandwidth contention between the guest and Pontoon's migration threads.

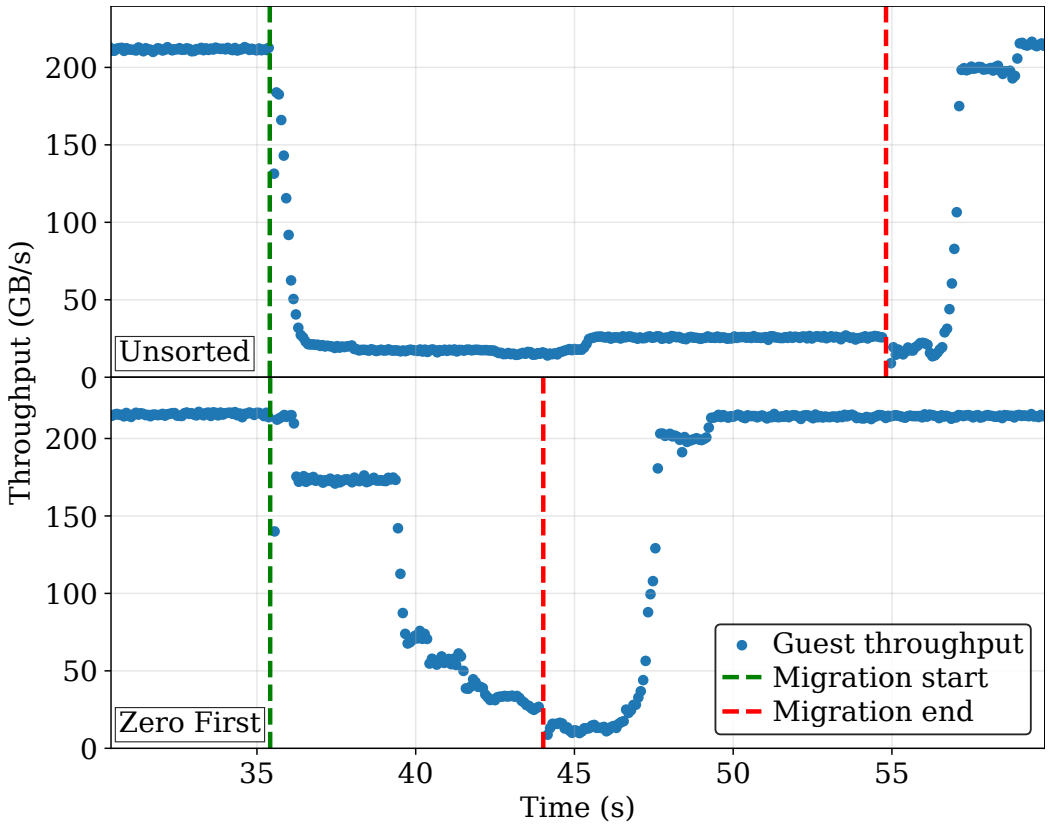


Figure 6.6: Zero-page-first demotion on a high-throughput 2 GB memory-churn working set placed early in address order, making unsorted demotion unfavorable. Zero-first demotion shortens migration time from 19.4 s to 8.6 s, reduces lost work during migration by 74.5%, and improves migration efficiency from 12.4% to 50.6%.

We constructed a worst-case placement for address-ordered demotion using the synthetic memory-churn workload: a 2 GB working set in a 64 GB VM with high throughput. We placed the working set near the start of guest memory, so

UNSORTED demoted it to CXL early, sharing CXL bandwidth with migration traffic. ZERO FIRST delayed demoting the active region by processing zero pages first.

We quantified application impact over the migration window using two metrics. *Lost work* is the area between the throughput baseline and the measured throughput curve from migration start to end. *Migration efficiency* is the fraction of baseline work completed during migration.

ZERO FIRST improved migration time and throughput during migration. Migration time fell from 19.4 s to 8.6 s, or 55.7%. Mean application throughput during migration increased from 26.3 GB/s to 109.3 GB/s. As a result, lost work during migration decreased from 3598.4 GB to 916.2 GB, or 74.5%, and migration efficiency improved from 12.4% to 50.6%.

These results also reflect a highly favorable case rather than a typical benefit. The advantage of ZERO FIRST depends on how many zero pages exist, where the working set falls in the demotion order, how bandwidth-intensive the workload is, and how sensitive it is to CXL contention.

6.4.7 Pushing CXL's limits

Figure 6.7 shows Pontoon's behavior under increasing synthetic memory-churn workloads. Each plot shows guest throughput before, during, and after CXL-based migration, with the migration window marked by green and red dashed lines. These tests went far beyond the dirtying rates where network-based migration was possible: TCP and RDMA convergence failed under only a small fraction of this workload.

The first two configurations used a 64 GB VM with similar pre-migration throughput, but increased the workload threads from 16 to 32. The final configuration scaled to a 232 GB VM with 200 vCPUs and 200 workload threads (the largest our experimental setup could handle).

We quantified both migration-window brownout and post-migration recovery. Again, *migration lost work* is the area between the pre-migration throughput baseline and the measured throughput curve during migration. For post-migration recovery, *recovery lost work* is the area between the throughput baseline and the measured throughput curve until the workload returned to stable performance.

The first two plots show that CXL was not primarily limited by the guest's aggregate throughput. Both tests sustained roughly 185 GB/s before migration and about 66 GB/s on average during migration. However, raising the workload

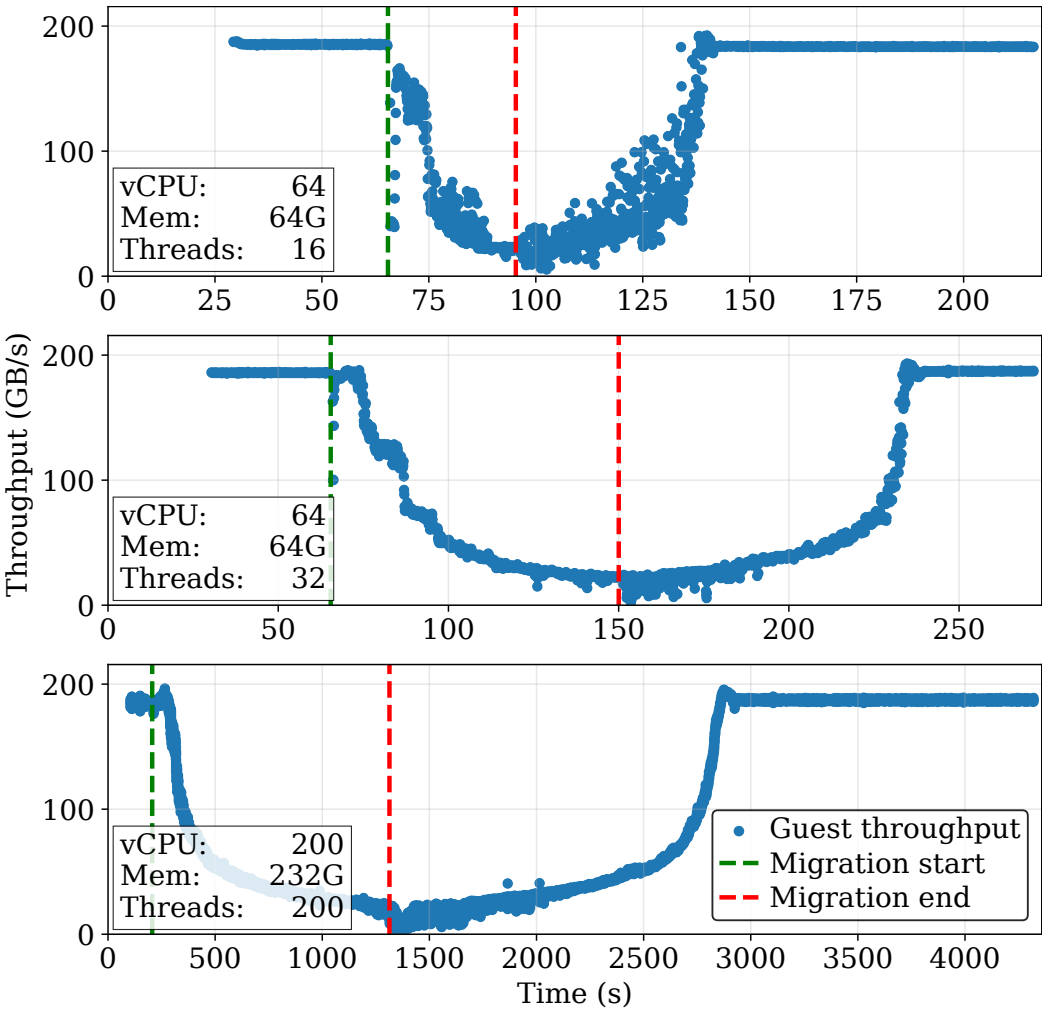


Figure 6.7: CXL stress test under increasingly threaded memory-churn workloads and larger VMs. As thread count and VM scale increase, CXL bandwidth contention stretches Pontoon's migration. At the largest scale, Pontoon still completes but takes 1107.5 s to migrate and 1545.1 s to recover.

from 16 to 32 threads increased migration time from 29.9 s to 84.6 s, while post-migration recovery increased from 38.5 s to 83.7 s. Thus, the brownout depth was similar, but the system remained in that degraded state for much longer.

The bottom plot shows the same effect at a larger scale. With a 232 GB VM and 200 workload threads, Pontoon still completed migration, but migration time increased to 1107.5 s and post-migration recovery took 1545.1 s. During migration, mean guest throughput remained 56.3 GB/s, corresponding to 30.5% migration efficiency. After migration, recovery efficiency was 22.7%, similar to the smaller configurations, but the recovery interval was much longer.

These results identify the dominant bottleneck in our CXL implementation. All configurations eventually completed, but extreme thread counts exposed CXL bandwidth contention as the dominant cost. This contention stretched both demotion and promotion, producing extended brownout rather than a convergence failure. Migration and recovery finished, but the guest spent a long time sharing CXL bandwidth with Pontoon's migration and promotion threads.

6.4.8 Blackout Time Analysis

We instrumented migration to identify the sources of blackout time. Pontoon uses `WBNOINVD` when available, which writes back dirty cache lines without invalidating them, and otherwise falls back to `WBINVD`. The latency of these instructions depends on the LLC size and the amount of dirty cache at flush time. On our system, with a 32 MB LLC, we observed idle-guest flush times as low as 0.4 ms and up to 7.2 ms under high-bandwidth workloads. If neither instruction is available, we fall back to iterative `CLFLUSHOPT`, whose cost scales with dirty memory size and can exceed 100 ms for large dirty sets.

For CXL, the vast majority of the observed blackout time came from fixed QEMU work, including CPU and device-state serialization, VM teardown, and migration cleanup. For example, the 29 ms blackout in Table 6.2 spent only 0.4 ms in `WBNOINVD`, while the remaining time came from this work.

This observation led us to identify QEMU's `virtio-blk` `ioeventfd` synchronization as the dominant component of blackout in our experiments. As an experimental optimization, we added support for disabling these before blackout. With an idle 8-vCPU guest, this reduced CXL blackout from 29 ms to 14 ms. With 64 vCPUs and no guest workload, pre-stopping `ioeventfd`s dropped blackout from 136 ms to 25 ms. With 64 vCPUs and a high-bandwidth workload, the same optimization cut it from 175 ms to 39 ms.

This synchronization overhead is also present in TCP and RDMA migration,

but is masked by network-based stop-and-copy's dirty-page transfer during blackout. CXL's much lower blackout time exposes these costs more clearly, making them both easier to identify and more worthwhile to optimize. We suspect additional optimizations are possible, but leave a broader redesign of this phase to future work.

6.5 Related Work

Pre-copy and post-copy migration. Clark *et al.* introduced pre-copy live migration [114, 33], which iteratively copies memory; Google reported 50 ms median and 300 ms 99th-percentile blackout across its production fleet [122]. Many optimizations improve pre-copy: adaptive stop-and-copy thresholds [71], ML-based termination predictors [61], copy-on-write preservation [45], and multi-core parallelization [130]—but all still retransmit dirtied pages and cannot eliminate this fundamental overhead. Post-copy [64] transfers each page at most once and avoids pre-copy's convergence problem, but exposes VMs to demand-paging latency after switchover. Hybrid approaches begin with pre-copy and can therefore retransmit dirty pages before switching to post-copy, after which remaining pages are fetched on demand. Pontoon avoids retransmission by demoting each DRAM-resident page to shared CXL memory once and remapping it so subsequent writes update the shared location directly; it avoids post-copy's latency penalty because the destination can already access the pages.

RDMA-based migration. RDMA dramatically reduces migration time compared to TCP [68, 74], and later work migrates SR-IOV InfiniBand virtual functions directly to preserve device access when migrating the VM [56]. However, RDMA still copies over the network and requires complex device-state coordination. Pontoon uses memory-mapped CXL rather than the network, eliminating network transfer of guest memory.

Shared-memory migration. ThymesisFlow uses IBM POWER9 disaggregated memory to leave guest memory in place, improving guest-visible memory latency and throughput during migration by up to three orders of magnitude while matching the best-case blackout of traditional approaches [55]; migration over software distributed shared memory reduces total migration time by 70% [77]. These works show that shared memory can reduce migration overhead, but rely on non-CXL or software-distributed shared memory mechanisms. Closest to Pontoon, a 2023

Linux summit BoF discussed CXL-based migration [132], but did not release an implementation. In contrast, Pontoon targets commodity x86 servers with standard CXL support and provides a working QEMU/KVM implementation.

CXL memory tiering. A substantial body of work addresses tiering policies for CXL memory. TPP extended Linux NUMA balancing to demote cold pages and promote hot ones [100]. Nomad used shadow copies and transactional migration to reduce thrashing [160]. Alto suppressed unnecessary migrations using memory-level parallelism awareness [93]. Colloid steers placement using each tier's loaded access latency [149], and HybridTier tracks long-term access frequency alongside recent momentum with compact probabilistic metadata [129]. Melody showed that some workloads tolerate CXL-only placement with little slowdown [92]. Pontoon is complementary to this work: tiering policies determine which pages reside in CXL during normal operation, and Pontoon leverages that placement during migration—pages already in shared CXL require no demotion at all.

CXL memory pooling. POND demonstrated CXL-based memory pooling, showing that pooling across 8–16 sockets captures most of the benefits [86]. Pontoon is complementary: VMs can migrate while leaving memory in a CXL pool, further reducing migration overhead.

6.6 Chapter Conclusion

We presented Pontoon, a live VM migration system that leverages CXL shared memory to eliminate fundamental inefficiencies of traditional migration. By reframing migration as a memory-placement problem—single-pass demotion to shared CXL, stop-and-flush, background promotion at the destination—Pontoon transfers each page at most once and avoids both pre-copy's retransmission and post-copy's demand-paging latency.

Our evaluation on CXL pooling devices showed dramatic improvements across all metrics. Pontoon completed 64 GB VM migrations in roughly 4.9 s, which is 2.1–9.4$\times$ faster than TCP, RDMA, and RDMA-pinall migration. Blackout times dropped from 267–281 ms to 29 ms, significantly reducing one of the most disruptive components of live migration for interactive services. Crucially, Pontoon's performance remains consistent up to dirty-page rates where network-

based migration fails to converge, eliminating the convergence problem that has plagued live migration for two decades.

Chapter 7

Conclusions

It is our thesis that understanding and optimizing tiered memory and storage systems requires comprehensive evaluation of multi-tier configurations, efficient exploration of configuration spaces, and novel tiering mechanisms that exploit opportunities introduced by evolving technologies. The configuration space of tiered memory and storage systems continues to grow as new devices, policies, and parameters are introduced, making these systems increasingly difficult to evaluate and explore efficiently. Emerging technologies such as Compute Express Link (CXL) further expand this space while enabling new forms of tiering.

Chapter 4 addresses the challenge of comprehensively evaluating tiered memory and storage systems. It examines limitations in existing cache analysis and presents a general n-level cache simulator. The simulator models arbitrary hierarchies and reports both per-tier and aggregate metrics, including performance and cost. Simulations using real-world traces show that conclusions about device selection, hierarchy complexity, and write policy can change when complete multi-tier configurations are evaluated across multiple metrics.

Chapter 5 addresses the challenge of efficiently exploring large configuration spaces. It presents a framework that combines hash-based sampling, Ramer-Douglas-Peucker curve simplification, recursive knee detection, and post-processing filters to select key points from miss ratio curves. The framework includes Z-Method, which uses statistical outlier detection to identify multiple significant knees. It identified optimal two-tier configurations while reducing the required simulations by average factors of 5.5× for ARC and 7.7× for LRU, and accelerated evolutionary optimization by 34%.

Chapter 6 addresses the challenge of exploiting tiering opportunities introduced by evolving technologies. It presents Pontoon, a rack-local fast path for

live virtual machine migration using shared CXL memory. Pontoon is implemented in QEMU/KVM and provides transparent guest memory tiering across local DRAM and shared CXL memory, performing migration in three phases: single-pass demotion of pages from DRAM to CXL, a brief stop-and-flush phase, and background promotion at the destination. For 64 GB VMs, Pontoon improved migration time by 2.1–9.4×, reduced blackout from 267–281 ms to 29 ms, and completed migration at dirty-page rates where traditional pre-copy failed to converge.

Together, these contributions provide techniques for evaluating multi-tier configurations, reducing the cost of searching their configuration spaces, and exploiting new tiering opportunities. The simulator reveals interactions and trade-offs that narrower analyses can miss, the key point selection framework efficiently identifies promising configurations, and Pontoon uses shared CXL memory to reduce migration time, blackout, and data movement during live VM migration.

7.1 Future Work

The placement of guest memory pages across local DRAM and shared CXL memory affects two important outcomes: guest performance and migration cost. Pages in DRAM provide lower access latency for the guest, while pages already resident in CXL do not need to be transferred during migration. These observations motivate two directions for future work: developing migration-aware CXL tiering policies and extending VM scheduling for CXL.

7.1.1 Migration-Aware CXL Tiering

Pontoon provides a transparent guest memory tiering mechanism that places and moves pages across local DRAM and shared CXL memory. Its current tiering policy, however, does not dynamically classify pages or adjust their placement during normal execution. Existing CXL tiering policies use signals such as access frequency, recency, loaded access latency, and memory level parallelism to identify which pages should remain in DRAM and which can be moved to CXL memory [100, 149, 93, 129]. An important direction for future work is to determine how these policies can operate alongside Pontoon and coordinate their page movements with migration.

Adapting existing tiering policies. Tiering approaches operate at different levels of the system and rely on different mechanisms for tracking memory accesses and moving pages. TPP implements page tracking and migration within the Linux kernel, while Nomad implements a Linux kernel mechanism for transactional page migration and page shadowing [100, 160]. HybridTier runs in user space and relies on hardware access sampling and operating system support, while Colloid and Alto modify the placement decisions made by other tiering systems [129, 149, 93]. Because Pontoon manages guest memory from user space within QEMU/KVM, these approaches may require substantial changes before their tiering mechanisms can be used in conjunction with Pontoon. Future work should determine which policies are compatible, what access information and system support they require, and how their placement decisions can be enforced. Their page movements must also be coordinated with migration, since promotions and demotions could interfere while Pontoon protects, copies, and remaps pages. Page movement may therefore need to be suspended during migration, while access tracking may continue so that the resulting information can guide tiering decisions at the destination.

Coordinating tiering with migration planning. Tiering policies typically make decisions based on page access behavior and memory pressure, but they lack information about VM migrations. A VM scheduler knows that a migration is planned, its expected timing, possible destinations, and the CXL resources available before it begins. This information could allow a tiering policy to prepare the VM by opportunistically moving pages to shared CXL memory before the active migration window. Future work should examine whether existing policies can be adapted to use this information or whether new policies are needed to jointly consider guest performance, page activity, expected migration cost, and shared CXL resource usage.

7.1.2 Extending VM Scheduling for CXL

Cloud VM schedulers assign VMs to hosts while satisfying resource requirements, fault domain constraints, and utilization goals [57]. CXL expands this problem by introducing additional decisions about the placement of guest memory and the use of shared memory for migration.

Scheduling with shared CXL resources. Traditionally, VM scheduling produces an assignment of VMs to hosts based on resource requirements and the resources available across the physical infrastructure. With shared CXL memory, the scheduler output may also need to specify how much memory each VM should place in local DRAM, which CXL pool it should use, and whether the selected placement supports future migration through that pool. These decisions are constrained by the CXL topology within each rack, since Pontoon can use shared memory only between hosts connected to the same pool. The scheduler must therefore consider which hosts share a CXL pool, along with its available capacity, fragmentation, and bandwidth, in addition to conventional host resources. It should also account for each VM's current page placement, since a VM with more pages in CXL may require less migration work. Future work should use these factors to select VMs and destinations, choose between Pontoon and TCP or RDMA, and coordinate concurrent migrations.

Improving VM packing through CXL migration. VM placements can become inefficient as VMs arrive, depart, and change their resource usage, stranding resources across hosts and preventing CPU, memory, and other resources from being fully utilized. Protean shows that live migration can be used to move VMs from fragmented machines and improve packing density and overall utilization [57]. Prior work on dynamic VM packing similarly shows that migration can improve the resulting placement, but that the benefit depends on the number and cost of the migrations performed [104]. Because Pontoon reduces migration time and blackout, it may allow a scheduler to correct inefficient placements more frequently and with less disruption to guests. This could make migration a more practical tool for ongoing load balancing rather than an operation reserved primarily for maintenance or severe imbalance. Future work should determine when the expected improvement in packing justifies migration, which VMs should be moved, and how frequently rebalancing should occur. These decisions must balance utilization gains against migration and recovery costs, temporary changes in guest performance, and contention for shared CXL bandwidth.

Bibliography

[1] AccuSim: Accurate simulation of cache replacement algorithms, March 2020.

[2] Charu C. Aggarwal. *Outlier Analysis*. Springer Publishing Company, Incorporated, 2nd edition, 2016.

[3] Jeffrey O. Agushaka, Absalom E. Ezugwu, Laith Abualigah, Samaher Khalaf Alharbi, and Hamiden Abd El-Wahed Khalifa. Efficient initialization methods for population-based metaheuristic algorithms: A comparative study. *Archives of Computational Methods in Engineering*, 30(3):1727–1787, April 2023.

[4] Waleed Ali, Sarina Sulaiman, and Norbahiah Ahmad. Performance improvement of least-recently-used policy in web proxy cache replacement using supervised machine learning. In *SOCO*, 2014.

[5] Amazon Web Services. Amazon EC2 instance types, 2025. Accessed: 2025-12-09.

[6] Amazon Web Services. Amazon EC2 U7i instances, 2025. Up to 32 TB memory per instance.

[7] Anandtech: Hardware news and tech reviews since 1997. *www.anandtech.com*.

[8] Mário Antunes, Henrique Aguiar, and Diogo Gomes. AL and S methods: Two extensions for L-method. In *7th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 371–376. IEEE, 2019.

[9] Mário Antunes, Diogo Gomes, and Rui L Aguiar. Knee/elbow estimation based on first derivative threshold. In *Fourth IEEE International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 237–240, Bamberg, Germany, March 2018. IEEE.

[10] Mário Antunes, Joana Ribeiro, Diogo Gomes, and Rui L Aguiar. Knee/elbow point estimation through thresholding. In *6th IEEE International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 413–419, Barcelona, Spain, August 2018. IEEE.

[11] Dulcardo Arteaga, Jorge Cabrera-Gámez, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. CloudCache: On-demand flash cache management for cloud computing. In *USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[12] Adnan Ashraf, Sobia Pervaiz, Waqas Bangyal, Kashif Nisar, Ag Asri Ag Ibrahim, Joel Rodrigues, and Danda Rawat. Studying the impact of initialization for population-based algorithms with low-discrepancy sequences. *Applied Sciences*, 11:8190, 09 2021.

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

[14] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75, 2015.

[15] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching — dynamic reallocation from cache-rich to cache-poor. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[16] Daniel S. Berger, Karthik Kumar, Midhul Vuppalapati, Chet Douglas, Jesse Sathre, Ian N. Robinson, Prateek Tandon, and Mark D. Hill. CXL in cloud practice: Practical lessons for incrementally scaling deployment. *IEEE Transactions on Computers*, 75(4):1234–1246, April 2026.

[17] Ricardo Bianchini, Marcus Fontoura, Eli Cortez, Anand Bonde, Alexandre Muzio, Ana-Maria Constantin, Thomas Moscibroda, Gabriel Magalhaes,

Girish Bablani, and Mark Russinovich. Toward ML-centric cloud platforms. *Communications of the ACM*, 63(2):50–59, 2020.

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[19] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: Miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 84–95, Philadelphia, PA, June 2018.

[20] Yongtao Cao, Byran J. Smucker, and Timothy J. Robinson. On using the hypervolume indicator to compare Pareto fronts: Applications to multi-criteria optimal experimental design. *Journal of Statistical Planning and Inference*, 160:60–74, 2015.

[21] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *USENIX Annual Technical Conference, (ATC)*, pages 893–907, Boston, MA, July 2018.

[22] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS'16, pages 323–336, New York, NY, USA, 2016. ACM.

[23] Xian Chen, Wenzhi Chen, Zhongyong Lu, Peng Long, Shuiqiao Yang, and Zonghui Wang. A duplication-aware SSD-based cache architecture for primary storage in virtualization environment. *IEEE Systems Journal*, 11(4):2578–2589, December 2017.

[24] Xunchao Chen, Navid Khoshavi, Jian Zhou, Dan Huang, Ronald F. DeMara, Jun Wang, Wujie Wen, and Yiran Chen. AOS: Adaptive overwrite scheme for energy-efficient MLC STT-RAM cache. In *2016 53rd*

ACM/EDAC/IEEE Design Automation Conference (DAC), pages 170:1–170:6, June 2016.

[25] Lerong Cheng, Jinjun Xiong, and Lei He. Non-Gaussian statistical timing analysis using second-order polynomial fitting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(1):130–140, 2008.

[26] Yue Cheng, Aayush Gupta, Anna Povzner, and Ali R. Butt. High performance in-memory caching through flexible fine-grained services. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Yuxia Cheng, Wenzhi Chen, Zonghui Wang, Xinjie Yu, and Yang Xiang. AMC: an adaptive multi-level cache algorithm in hybrid storage systems. *Concurrency and Computation: Practice and Experience*, 27(16):4230–4246, 2015.

[28] Yuxia Cheng, Yang Xiang, Wenzhi Chen, Houcine Hassan, and Abdulhameed Alelaiwi. Efficient cache resource aggregation using adaptive multi-level exclusive caching policies. *Future Generation Computer Systems*, 86:964–974, 2018.

[29] Davide Chicco and Giuseppe Jurman. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21, 2020.

[30] Anita Choudhary, Mahesh Chandra Govil, Girdhari Singh, Lalit K. Awasthi, Emmanuel S. Pilli, and Divya Kapil. A critical survey of live virtual machine migration techniques. *J. Cloud Comput.*, 6(1), December 2017.

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

[32] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.

[33] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, USA, 2005. USENIX Association.

[34] Compute Express Link Consortium. Compute express link (CXL). *https://computeexpresslink.org/*, 2025. Accessed: 2025-07-27.

[35] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, Shanghai, China, 2017.

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

[37] Axel de Perthuis de Laillevault, Benjamin Doerr, and Carola Doerr. Money for nothing: Speeding up evolutionary algorithms through better initialization. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15. ACM, July 2015.

[38] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[39] Dinero IV trace-driven uniprocessor cache simulator. *http://pages.cs.wisc.edu/~markhill/DineroIV/*.

[40] Nosayba El-Sayed, Ioan A. Stefanovici, George Amvrosiadis, Andy A. Hwang, and Bianca Schroeder. Temperature management in data centers: Why some (might) like it hot. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'12, pages 163–174, New York, NY, USA, 2012. ACM.

[41] Tyler Estro. mtc-sim: Multi-tier cache simulator, 2026. *https://github.com/tyestro/mtc-sim*.

[42] Tyler Estro, Mário Antunes, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. Guiding simulations of multi-tier storage caches using knee detection. In *31st Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '23)*, Stony Brook, NY, October 2023. IEEE Computer Society.

[43] Tyler Estro, Mário Antunes, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. Accelerating multi-tier storage cache simulations using knee detection. *Performance Evaluation*, 164:102410, May 2024.

[44] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. Desperately seeking ... optimal multi-tier cache configurations. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage (HotStorage '20)*, Boston, MA, July 2020. USENIX.

[45] Roja Eswaran, Mingjie Yan, and Kartik Gopalan. Template-aware live migration of virtual machines. In *Proceedings of the Eighth ACM/IEEE Symposium on Edge Computing*, SEC '23, pages 336–340, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, 2012.

[47] Dinuni Fernando, Jonathan Terner, Kartik Gopalan, and Ping Yang. Live migration ate my vm: Recovering a virtual machine after failure of post-copy live migration. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 343–351, 2019.

[48] Diogo Freitas, Luiz Guerreiro Lopes, and Fernando Morgado Dias. Particle swarm optimisation: A historical review up to the current developments. *Entropy*, 22, 2020.

[49] Jianyu Fu, Dulcardo Arteaga, and Ming Zhao. Locality-driven MRC construction and cache allocation. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 19–20, New York, NY, USA, 2018. ACM.

[50] M. García-Arnau, D. Manrique, J. Ríos, and A. Rodríguez-Patón. Initialization method for grammar-guided genetic programming. *Knowledge-Based Systems*, 20(2):127–133, 2007. AI 2006.

[51] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[52] Ron Goldman. Curvature formulas for implicit curves and surfaces. *Computer Aided Geometric Design*, 22(7):632–658, October 2005.

[53] Google Cloud. Choose a disk type — compute engine disks, 2025. Accessed: 2025-12-09.

[54] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[55] Andreas Grapentin, Felix Eberhardt, Tobias Zagorni, Andreas Polze, Michele Gazzetti, and Christian Pinto. Zero-Copy, Minimal-Blackout Virtual Machine Migrations Using Disaggregated Shared Memory. In João Bispo, Sotirios Xydis, Serena Curzel, and Luís Miguel Sousa, editors, *15th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 13th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2024)*, volume 116 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:13, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[56] Wei Lin Guay, Sven-Arne Reinemo, Bjørn Dag Johnsen, Chien-Hua Yen, Tor Skeie, Olav Lysne, and Ola Tørudbakken. Early experiences with live migration of SR-IOV enabled InfiniBand. *Journal of Parallel and Distributed Computing*, 78:39–52, 2015.

[57] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861. USENIX Association, November 2020.

[58] Ubaid Ullah Hafeez, Muhammad Wajahat, and Anshul Gandhi. ElMem: Towards an elastic memcached system. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems*, pages 278–289, Vienna, Austria, 2018.

[59] Alireza Haghdoost. Sim-ideal, Dec 2013. *https://github.com/arh/sim-ideal/tree/master*.

[60] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 161–175, New York, NY, USA, 2015. ACM.

[61] Raseena M. Haris, Mahmoud Barhamgi, Armstrong Nhlabatsi, and Khaled M. Khan. Optimizing pre-copy live virtual machine migration in cloud computing using machine learning-based prediction model. *Computing*, 106(9):3031–3062, July 2024.

[62] Lulu He, Zhibin Yu, and Hai Jin. FractalMRC: Online cache miss rate curve prediction on commodity systems. In *IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1341–1351, 2012.

[63] Tianzhang He and Rajkumar Buyya. A taxonomy of live migration management in cloud computing. *ACM Computing Surveys*, 56(3):56:1–56:33, March 2024.

[64] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 43(3):14–26, July 2009.

[65] John H. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, 1992.

[66] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 351–364, Denver, CO, June 2016. USENIX Association.

[67] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Transactions on Storage (TOS)*, 14:12:1–12:34, 2018.

[68] Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K. Panda. High performance virtual machine migration with RDMA over modern interconnects. In *2007 IEEE International Conference on Cluster Computing*, pages 11–20, 2007.

[69] Wei Huang, Jiuxing Liu, Matthew Koop, Bulent Abali, and Dhabaleswar Panda. Nomad: migrating OS-bypass networks in virtual machines. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 158–168, New York, NY, USA, 2007. Association for Computing Machinery.

[70] Elyse Ge Hylander. Azure delivers the first cloud VM with Intel Xeon 6 and CXL memory - now in private preview, November 2025. Accessed: 2025-12-10.

[71] Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[72] Muhammad Imran, Muhammad Ibrahim, Muhammad Salah Ud Din, Muhammad Atif Ur Rehman, and Byung Seo Kim. Live virtual machine migration: A survey, research challenges, and future directions. *Computers and Electrical Engineering*, 103:108297, 2022.

[73] Intel Corporation. Page Modification Logging for Virtual Machine Monitor. White Paper 331872-001, Intel Corporation, January 2015.

[74] C. Isci, J. Liu, B. Abali, J. O. Kephart, and J. Kouloheris. Improving server utilization using fast virtual machine migration. *IBM Journal of Research and Development*, 55(6):4:1–4:12, 2011.

[75] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR'14, pages 253–262, New York, NY, USA, 2014. ACM.

[76] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. The pitfalls of deploying solid-state drive RAIDs. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 14:1–14:13, Haifa, Israel, June 2011. Association for Computing Machinery.

[77] Changyeon Jo, Hyunik Kim, and Bernhard Egger. Instant virtual machine live migration. In Karim Djemame, Jörn Altmann, José Ángel Bañares, Orna Agmon Ben-Yehuda, Vlado Stankovski, and Bruno Tuffin, editors, *Economics of Grids, Clouds, Systems, and Services*, pages 155–170, Cham, 2020. Springer International Publishing.

[78] M. Jung and M. Kandemir. Revisiting widely held SSD expectations and rethinking system-level implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 203–216, New York, NY, USA, 2013. ACM.

[79] Borhan Kazimipour, Xiaodong Li, and A. K. Qin. A review of population initialization techniques for evolutionary algorithms. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2585–2592, 2014.

[80] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995.

[81] Ricardo Koller, Akshat Verma, and Raju Rangaswami. Generalized ERSS tree model: Revisiting working sets. *Performance Evaluation*, 67:1139–1154, 2010.

[82] Iwona Kotlarska, Andrzej Jackowski, Krzysztof Lichota, Michal Welnicki, Cezary Dubnicki, and Konrad Iwanicki. InftyDedup: Scalable and Cost-Effective cloud tiering with deduplication. In *21st USENIX Conference on*

File and Storage Technologies (FAST 23), pages 33–48, Santa Clara, CA, February 2023. USENIX Association.

[83] Saku Kukkonen and Jouni Lampinen. Gde3: The third evolution step of generalized differential evolution. In *2005 IEEE congress on evolutionary computation*, volume 1, pages 443–450. IEEE, 2005.

[84] Jean-Christophe Léger. Menger curvature and rectifiability. *Annals of Mathematics*, 149(3):831–869, 1999.

[85] Sebastien Levy, Randolph Yao, Youjiang Wu, Yingnong Dang, Peng Huang, Zheng Mu, Pu Zhao, Tarun Ramani, Naga Govindaraju, Xukun Li, Qingwei Lin, Gil Lapid Shafriri, and Murali Chintalapati. Predictive and adaptive failure mitigation to avert production cloud VM interruptions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1155–1170. USENIX Association, 2020.

[86] 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, pages 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[87] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[88] Miqing Li and Xin Yao. Quality evaluation of solution sets in multiobjective optimisation: A survey. *ACM Comput. Surv.*, 52(2), March 2019.

[89] Z. Li, M. Chen, A. Mukker, and E. Zadok. On the trade-offs among performance, energy, and endurance in a versatile hybrid drive. *ACM Transactions on Storage (TOS)*, 11(3), July 2015.

[90] Z. Li, A. Mukker, and E. Zadok. On the importance of evaluating storage systems' $Costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[91] Chieh-Jan Mike Liang, Jie Liu, Liqian Luo, Andreas Terzis, and Feng Zhao. RACNet: A high-fidelity data center sensing network. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 15–28, New York, NY, USA, 2009. ACM.

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

[93] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered memory management beyond hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025.

[94] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. eMRC: Efficient miss rate approximation for multi-tier caching. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, February 2021.

[95] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San Jose, CA, February 2013. USENIX Association.

[96] Edson Ramiro Lucas Filho, Andreas Efstathiou, Lun Yang, Kebo Fu, Jianqiang Shen, and Herodotos Herodotou. DITIS: An end-to-end system-level simulator and optimizer for distributed tiered storage. *SN Computer Science*, 6(6), August 2025.

[97] Edson Ramiro Lucas Filho, Lambros Odysseos, Yang Lun, Fu Kebo, and Herodotos Herodotou. DITIS: A distributed tiered storage simulator. *Infocommunications Journal*, XIV(4):18–25, December 2022.

[98] Zhongkun Ma and Guy A. E. Vandenbosch. Impact of random number generators on the performance of particle swarm optimization in antenna design. In *2012 6th European Conference on Antennas and Propagation (EUCAP)*, pages 925–929, 2012.

[99] Rano Mal and Yul Chu. A flexible multi-core functional cache simulator (FM-SIM). In *Proceedings of the Summer Simulation Multi-Conference*, SummerSim '17, San Diego, CA, USA, 2017. Society for Computer Simulation International.

[100] 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, pages 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[101] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. XvMotion: Unified virtual machine migration over long distance. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 97–108, Philadelphia, PA, June 2014. USENIX Association.

[102] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[103] Nimrod Megiddo and Dharmendra Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '03)*, pages 115–130, San Francisco, CA, March 2003. USENIX Association.

[104] Konstantina Mellou, Marco Molinaro, and Rudy Zhou. The power of migrations in dynamic bin packing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(3):45:1–45:28, 2024.

[105] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, New York, NY, USA, 2011. ACM.

[106] Microsoft Azure. Predicting and recovering from virtual machine failures with resiliency, September 2018.

[107] Microsoft Azure. Mdsv3 very high memory series — memory optimized virtual machines, 2025. Up to 31 TB memory per VM.

[108] Microsoft Corporation. Azure VM sizes with no local temporary disk, 2025. Accessed: 2025-12-09.

[109] Shaily Mittal and Nitin. Memory map: A multiprocessor cache simulator. *Journal of Electrical and Computer Engineering*, 2012:365091:1–365091:12, September 2012.

[110] Seyed Jalaleddin Mousavirad, Azam Asilian Bidgoli, and Shahryar Rahnamayan. Tackling deceptive optimization problems using opposition-based DE with center-based Latin hypercube initialization. In *2019 14th International Conference on Computer Science & Education (ICCSE)*, pages 394–400, 2019.

[111] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 2008)*, 2008.

[112] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace set 388). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2007.

[113] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? when? and why? In *Proceedings of the Ninth ACM Israeli Experimental Systems Conference (SYSTOR '16)*, pages 7:1–7:11, Haifa, Israel, May 2016. ACM.

[114] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[115] Anant Vithal Nori, Jayesh Gaur, Siddarth Rai, Sreenivas Subramoney, and Hong Wang. Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies. In *45th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 96–109, June 2018.

[116] Massachusetts Institute of Technology. DynamoRIO: Dynamic instrumentation tool platform, February 2009. *http://www.dynamorio.org/*.

[117] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, 2019.

[118] QEMU Project. QEMU migration documentation. *https://www.qemu.org/docs/master/devel/migration/main.html*, 2026. Accessed July 27, 2026.

[119] Shahryar Rahnamayan, Hamid R. Tizhoosh, and Magdy M.A. Salama. A novel population initialization method for accelerating evolutionary algorithms. *Computers & Mathematics with Applications*, 53(10):1605–1614, 2007.

[120] Sundaresan Rajasekaran, Shaohua Duan, Wei Zhang, and Timothy Wood. Multi-cache: Dynamic, efficient partitioning for multi-tier caches in consolidated VM environments. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 182–191. IEEE, April 2016.

[121] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244–256, 1972.

[122] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. VM live migration at scale. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18, pages 45–56, New York, NY, USA, 2018. Association for Computing Machinery.

[123] R. Salkhordeh, S. Ebrahimi, and H. Asadi. Reca: An efficient reconfigurable cache architecture for storage systems with online workload characterization. *IEEE Transactions on Parallel and Distributed Systems*, 29(7):1605–1620, July 2018.

[124] Stan Salvador and Philip Chan. Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In *16th IEEE*

International Conference on Tools With Artificial Intelligence, pages 576–584. IEEE, 2004.

[125] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To ARC or not to ARC. In *HotStorage*, 2015.

[126] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, Minneapolis, MN, June 2011. IEEE.

[127] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating performance and energy in file system server workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '10)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[128] Kia Shakiba, Sari Sultan, and Michael Stumm. Kosmo: Efficient online miss ratio curve generation for eviction policy evaluation. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 89–105, Santa Clara, CA, February 2024. USENIX Association.

[129] Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. HybridTier: An adaptive and lightweight CXL-memory tiering system. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 112–128. Association for Computing Machinery, 2025.

[130] Xiang Song, Jicheng Shi, Ran Liu, Jian Yang, and Haibo Chen. Parallelizing live migration of virtual machines. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 85–96, New York, NY, USA, 2013. Association for Computing Machinery.

[131] Carl Staelin and Hector Garcia-Molina. Clustering active disk data to improve disk performance. Technical Report TR-283-90, Princeton University, NJ, USA, 1990.

[132] Dragan Stancevic. nilmigration: Nearly instantaneous live migration of virtual machines over cxl. *https://nil-migration.org/*, September 2022. BoF pre-

sented at Linux Storage, Filesystem, Memory Management & BPF Summit, May 2023.

[133] R. Storn. On the usage of differential evolution for function optimization. In *Proceedings of North American Fuzzy Information Processing*, pages 519–523, 1996.

[134] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 513–527, Berkeley, CA, USA, 2015. USENIX Association.

[135] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. *ACM Sigplan Notices*, 44(3):121–132, 2009.

[136] K.C. Tan, T.H. Lee, and E.F. Khor. Evolutionary algorithms for multi-objective optimization: performance assessments and comparisons. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, volume 2, pages 979–986 vol. 2, 2001.

[137] Elvira Teran, Zhe Wang, and Daniel A. Jiménez. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[138] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*, pages 1–14. ACM, 2020.

[139] Xavier Tolsa. Principal values for the cauchy integral and rectifiability. *Proceedings of the American Mathematical Society*, 128(7):2111–2119, 2000.

[140] Thomas Tometzki and Sebastian Engell. Systematic initialization techniques for hybrid evolutionary algorithms for solving two-stage stochastic mixed-integer programs. *IEEE Transactions on Evolutionary Computation*, 15:196–214, 2011.

[141] Tom's hardware: For the hardcore pc enthusiast. *www.tomshardware.com*.

[142] Anjul Tyagi, Tyler Estro, Geoff Kuenning, Erez Zadok, and Klaus Mueller. PC-Expo: A metrics-based interactive axes reordering method for parallel coordinate displays. *IEEE Transactions on Visualization and Computer Graphics*, October 2022.

[143] Musa Unal, Vishal Gupta, Yueyang Pan, Yujie Ren, and Sanidhya Kashyap. Tolerate it if you cannot reduce it: Handling latency in tiered memory. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, pages 50–57, New York, NY, USA, 2025. Association for Computing Machinery.

[144] UserBenchmark. *www.userbenchmark.com*.

[145] Nguyen Quang Uy, Nguyen Xuan Hoai, RI McKay, and Pham Minh Tuan. Initialising PSO with randomised low-discrepancy sequences: the comparative results. In *2007 IEEE Congress on Evolutionary Computation*, pages 1985–1992, 2007.

[146] A. Verma, R. Koller, L. Useche, and R. Rangaswami. SRCMap: Energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, 2010.

[147] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. FIU traces (SNIA IOTTA trace set 390). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2009.

[148] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage (HotStorage '18)*, Boston, MA, July 2018. USENIX.

[149] 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*, pages 79–94. ACM, 2024.

[150] Carl A. Waldspurger, Nohhyun Park, Alex Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015. USENIX Association.

[151] Carl A. Waldspurger, Trausti Saemundson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC '17)*, pages 487–498, Berkeley, CA, USA, 2017. USENIX Association.

[152] Han Wan, Xiaopeng Gao, Xiang Long, and Zhiqiang Wang. *GCSim: A GPU-Based Trace-Driven Simulator for Multi-level Cache*, pages 177–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[153] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, 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, pages 452–470. IEEE Press, 2024.

[154] Jiangtao Wang, Zhiliang Guo, and Xiaofeng Meng. An efficient design and implementation of multi-level cache for database systems. In *DASFAA*, 2015.

[155] Yun Wang, Liang Chen, Jie Ji, Xianting Tian, Ben Luo, Zhixiang Wei, Zhibai Huang, Kailiang Xu, Kaihuan Peng, Kaijie Guo, Ning Luo, Guangjian Wang, Shengdong Dai, Yibin Shen, Jiesheng Wu, and Zhengwei Qi. To PRI or not to PRI, that's the question. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[156] Marcel Weisgut, Daniel Ritter, Pinar Tözün, Lawrence Benson, and Tilmann Rabl. CXL memory performance for in-memory data processing. *Proceedings of the VLDB Endowment*, 18(9):3119–3133, May 2025.

[157] A. Wildani, E. L. Miller, and L. Ward. Efficiently identifying working sets in block I/O streams. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 5:1–5:12. ACM, 2011.

[158] John Wilkes. The Pantheon storage-system simulator, 1996.

[159] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks.

In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, Broomfield, CO, October 2014. USENIX Association.

[160] 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.

[161] Jun Xiao, Yaocheng Xiang, Xiaolin Wang, Yingwei Luo, Andy Pimentel, and Zhenlin Wang. Floria: A fast and featherlight approach for predicting cache performance. In *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, pages 25–36, New York, NY, USA, 2023. Association for Computing Machinery.

[162] Yizhe Xu, Yuan Tao, Zhibin Zhang, Kang Yan, Chao Zhang, Shuo Shi, Zongpu Zhang, Xu Huan, Yibin Shen, Xudong Zheng, Jiesheng Wu, Jian Li, and Haibing Guan. M3U: Scalable kernel memory management for efficient Post-Copy live migration of High-End virtual machines. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 1715–1730, Seattle, WA, July 2026. USENIX Association.

[163] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association.

[164] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[165] Juncheng Yang. PyMimircache. *https://github.com/1a1a11a/PyMimircache*. Retrieved April 17, 2019.

[166] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: Mining sporadic associations for cache prefetch-

ing. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 66–79, New York, NY, USA, 2017. ACM.

[167] Xinjun Yang, Qingda Hu, Junru Li, Feifei Li, Yicong Zhu, Yuqi Zhou, Qiuru Lin, Jian Dai, Yang Kong, Jiayu Zhang, Guoqiang Xu, and Qiang Liu. Beluga: A CXL-based memory architecture for scalable and efficient LLM KVCache management. *Proceedings of the ACM on Management of Data*, 4(1):1–29, April 2026.

[168] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. Unlocking the potential of CXL for disaggregated memory in cloud-native databases. In *Companion of the 2025 International Conference on Management of Data (SIGMOD/PODS '25)*, pages 689–702, New York, NY, USA, June 2025. Association for Computing Machinery. Industry Track Best Paper Award.

[169] Guo Yu, Lianbo Ma, Yaochu Jin, Wenli Du, Qiqi Liu, and Hengmin Zhang. A survey on knee-oriented multiobjective evolutionary optimization. *IEEE Transactions on Evolutionary Computation*, 26(6):1452–1472, December 2022.

[170] Guohui Zhang, Liang Gao, and Yang Shi. An effective genetic algorithm for the flexible job-shop scheduling problem. *Expert Systems with Applications*, 38(4):3563–3573, 2011.

[171] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '20, 2020.

[172] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based fault injection using selective state reset. In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, 2024. USENIX Association.

[173] Xinyu Zhang, Tyler Estro, Geoff Kuenning, Erez Zadok, and Klaus Mueller. Into the void: Mapping the unseen gaps in high dimensional data. *IEEE Transactions on Visualization & Computer Graphics*, 31:8578–8591, May 2025.

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

[175] Yuhong Zhong, Fiodar Kazhamiaka, Pantea Zardoshti, Shuwei Teng, Rodrigo Fonseca, Mark D. Hill, and Daniel S. Berger. Octopus: Enhancing CXL memory pods via sparse topology. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 1303–1322, Renton, WA, May 2026. USENIX Association.

[176] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving cash by using less cache. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'12, page 3, USA, 2012. USENIX Association.