Title Optimizing Block Storage Servers Using Multi-tier Caches

Author Pranav Bhandari

Degree Doctor of Philosophy

Program Computer Science

Approved by the Committee

Signed by: [illegible] 972660FCE384458...	Avani Wildani *Advisor*
Signed by: Vasily Tarasov 051DFF30F597418...	Vasily Tarasov *Committee Member*
Signed by: Ymir Vigfusson F3446880D847494...	Ymir Vigfusson *Committee Member*
Signed by: Michelangelo Grigni BC1D520389E1451...	Michelangelo Grigni *Committee Member*
	Committee Member
	Committee Member

Accepted by the Laney Graduate School:

Kimberly Jacob Arriola, Ph.D, MPH
Dean, James T. Laney Graduate School

Date

Optimizing Block Storage Servers Using Multi-tier Caches

By

Pranav Bhandari
B.S., Trinity College, 2012
M.S., Emory University, 2021

Advisor: Avani Wildani, Ph.D

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2024

Abstract

Optimizing Block Storage Servers Using Multi-tier Caches

By Pranav Bhandari

Storage systems persist large volumes of data and provide fast data access to applications that are ubiquitous in our society, such as banking, social networks, machine learning, video streaming, and ride hailing. The cheap, high-capacity, low-bandwidth backing store provides persistence, whereas the expensive, low-capacity, high-bandwidth cache delivers performance. Multiple storage nodes with backing store and cache provide the required capacity and performance. Large storage clusters with expensive hardware can be costly, both financially and ecologically. In order to reduce the size and consequently the cost of storage systems, we need to squeeze more performance from a single server. An approach is to add a flash cache to support the DRAM cache, which increases the potential throughput of a storage server. This can reduce the number of storage servers that are required to meet the performance requirement. However, it is challenging to determine when using a flash cache can be beneficial.

This dissertation compares the performance of storage servers with/without multi-tier caches using diverse servers and workloads, and develops techniques to determine when to use a multi-tier cache. First, we evaluate the potential performance and cost benefit of using multi-tier caches using simulation and analysis. We developed an algorithm, Cydonia, to determine cost-effective tier sizes given a workload and storage devices on the server. Next, we use trace replay to validate the performance improvement from multi-tier caches and demonstrate the importance of request rate along with miss ratio in determining performance. We train decision tree models that accurately predict whether using a multi-tier cache will improve performance using the large corpus of data collected using trace replay for a given server. We follow it up with BlkSample, a technique to generate accurate block trace samples that reduces the overhead of multi-tier cache analysis and replay.

Optimizing Block Storage Servers Using Multi-tier Caches

By

Pranav Bhandari
B.S., Trinity College, 2012
M.S., Emory University, 2021

Advisor: Avani Wildani, Ph.D

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2024

Acknowledgement

Through the ups and downs of the PhD journey, growth has been a constant. I am grateful for the unwavering support and encouragement of my family, mentors and friends. First, I would like to thank my adviser, Avani Wildani, for her positivity and encouragement, especially during challenging times. Thank you for always being within reach and mentoring me through life.

I would like to extend my gratitude to my thesis committee: Ymir Vigfusson, Michelangelo Grigni, and Vasily Tarasov, for their time and feedback. Each member has significantly impacted my PhD journey. Ymir has been a source of learning and inspiration throughout graduate school, and his pursuit of excellence is something that I aspire to emulate in my life. Prof. Grigni has been an exceptional teacher, whose passion for teaching and ability to break down complex information into simpler problems have made learning a pleasure. Vasily has taught me a lot about the industry. His insights gave me new perspectives that improved the quality of my work.

I have had the opportunity to collaborate with amazing people in industry and academia. Thanks to Vasily Tarasov, Lukas Rupprecht, Ali Anwar, Dimitrios Skourtis, Erez Zadok, Tyler Estro, Carl Waldspurger, Mário Antunes, Anshul Gandhi, and Geoff Kuenning. The people of Emory University, especially the SimBioSys research group, have been a source of joy. I am fortunate to know Reza Karimi, Derek Onken, Vishwanath Seshagiri, Juncheng Yang, Harshita Sahijwani, Zelalen Gero, Sapoonjyoti DuttaDuwarah, Lei Zhang, Sergio Gramacho, Gary Vestal, Yazhou Zhang, Shrey Gupta, and Si Chen.

Thank you to my family for prioritizing my education above everything. They have sacrificed a lot for me to be where I am today. I am grateful to my partner Keyuri Tatu. She has been part of the day to day grind on this journey. Her strength and optimism have helped me get to the finish line. Lastly, a special thanks to my cats, Mau and Coco, for their companionship.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Global digitization has replaced paper with storage devices such as hard disk drives (HDDs) and solid-state drives (SSDs) as the primary repository of human information. Storage systems utilize storage devices to store and retrieve information while providing high performance, reliability, availability, and scalability. Storage systems power digital assets such as social media applications, video streaming, payment processing, and government services, which are integral to the smooth functioning of the world. The pressure on storage systems continues to increase along with the volume of data that we generate, the number of users, and the stringent performance requirements needed to sustain new technologies. International Data Corporation forecasts that the amount of data created per year will continue to increase at a compound annual growth rate of 21.2% and will reach 221,000 exabytes by 2026 [45].

Storage servers typically use memory, a single-tier (ST) cache, to cache data persisted in the backing store. DRAM, being the fastest temporary storage device, is the most obvious candidate for the cache device, but it is expensive and limited in size. DRAM can be responsible for almost half the cost of a server in data centers, even when a server does not use the maximum amount or the highest grade DRAM [67, 5]. Servers that use GPU are the only servers where memory would not be the majority of the cost; however, most servers in the world do not contain a GPU. The limited size of DRAM means that to support large volumes of data, we have to scale horizontally by adding more servers. When you add more servers, you pay not only for more DRAM but also for all other components of the server. More servers also means higher complexity due to data management across multiple servers and higher operational cost due to increase in recurring costs for power, cooling, and teams required to support a complex storage infrastructure. Scaling vertically, while possible only up to a limit, is more cost-effective. Beefing up the servers as much as possible reduces the complexity and cost of the storage infrastructure as performance requirement can be met while using fewer servers. As there is only so much DRAM we can add to a server, one way to increase the cache size and increase the potential throughput of the server is to add a second tier of SSD cache to create a multi-tier (MT) cache. Figure 1.1 displays the traditional ST cache along with different types of MT caches. MT caches utilize a cheaper and larger SSD as an additional cache tier, increasing the cache size and lowering the miss ratio. A side effect of lowering the miss ratio is that we require fewer servers and less memory to access data at high speed, which means lower cost. Any performance optimization that we make can be viewed as a cost optimization problem.

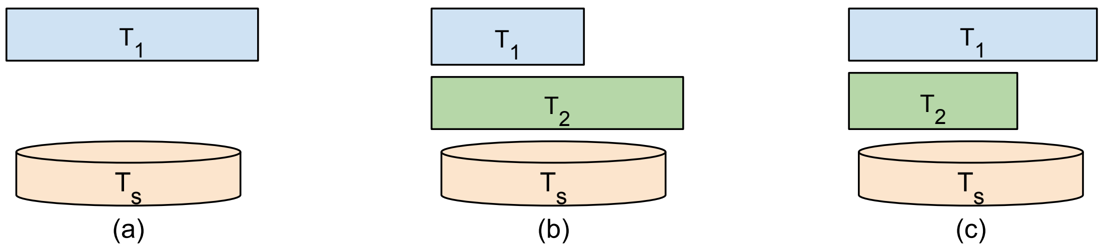


Figure 1.1: A single-tier cache (a) with a cache tier T_1 and a backing store T_S. A pyramidal (b) and non-pyramidal (c) multi-tier caches with two tiers T_1 and T_2 and a backing-store tier T_S.

We can try to reduce the cost of our cache by very carefully sizing it to meet the performance requirement. This approach ignores workload phase shifts and assumes a static workload. Caches are over-provisioned to ensure that, in the event of workload shifts, the performance does not degrade rapidly. Smaller caches are more susceptible to scans, as a small burst of new items which might never be reused again can clear the cache of all the warm and hot data. MT caches provide large cache capacity on a reasonable budget. However, there are many unknowns related to the effective use of MT caches and when it should not be used at all. Although the use of MT caches increases the effective cache size and the potential throughput of block storage systems, there is no guarantee that the actual throughput will improve. This is why we say *potential* throughput; it means that, for the correct workload and storage devices, a block storage system can deliver higher throughput with an MT cache than without it. It can be tempting to add a tier-2 cache in the hopes of improving the throughput and cost-efficiency of block storage systems without careful evaluation of the workload and storage devices already in a block storage server. If the additional overhead introduced by the tier-2 cache outweighs the gain from additional cache hits, the server throughput will be reduced even if the miss ratio is reduced. It is not trivial to estimate the overhead and gain of adding a tier-2 cache. Instances where no amounts of tier-2 cache can improve performance occur with non-negligible frequency. A better understanding of how workload and device properties interact to influence MT cache performance will help more storage system administrators safely deploy a tier-2 cache. MT caches will become more ubiquitous as we generate more data in the future, and optimizing the performance of the hardware and composing cost-efficient MT caches are important for making the technology more accessible and green. Our focus in this thesis is to find when adding a tier-2 cache can improve performance and develop efficient techniques to make this determination. In this thesis, we focus on block storage systems, but the findings of this thesis can be generalized to other storage systems such as key-value and file storage.

1.0.1 Thesis Scope

We define the cache policies that are constant throughout the thesis. The first is the replacement policy at each tier, which determines which requests are evicted when the cache is full. The next policy is the admission policy, which determines how blocks travel between caching tiers.

Storage System Storage systems apart from block storage like file and object storage are also used extensively. We focus on block storage systems as both file and object storage at the end of the day are stored as blocks at the low level. Our learning from optimizing for fixed-sized cache blocks can be translated to all storage systems.

Replacement Policy This thesis focuses on the most common cache replacement policy: Least Recently Used (LRU). LRU belongs to the family called *stack algorithms* [60], which satisfies a convenient *inclusion property*: for the same input sequence and replacement policy, all elements kept in a cache of size n are also included in caches with sizes greater than n.

Admission Policy The admission policy dictates actions on an eviction, miss, and hit; these, in turn, determine the request latency at each tier in an MT cache. In a two-tier *exclusive admission policy*, the most-recently-used blocks are admitted to the tier-1 cache, and blocks evicted from tier 1 are moved down to tier 2. In a two-tier *inclusive admission policy*, the blocks are admitted to both tiers, and thus the data blocks evicted from tier 1 do not have to be admitted to tier 2. In this thesis, we chose a two-tier exclusive admission policy for two reasons: (1) an inclusive admission policy is more expensive as blocks are duplicated in all tiers, making the overall cache less effective; and (2) we were interested in challenging conventional wisdom and explore cache hierarchies where the tier-2 cache is smaller than the tier-1 cache (non-pyramidal), but an inclusive admission policy makes this nearly impossible (*i.e.*, the tier-2 cache could never be smaller than tier 1, to ensure that all tier-1 items also reside in tier 2). We admit every admission from tier-1 cache to tier-2 cache, which can be ineffective and lead to flash write amplification [100].

1.1 Contributions

The thesis has two main goals. The first is to *determine when MT caches are effective and when not.* The second is to *develop efficient methods to determine* *when adding a tier-2 cache can improve performance.* We approach our first goal using trace simulation and replay of production traces on servers with different set of storage devices. We derive insights related to MT caching such as what kind of workload characteristics, storage device combinations favor MT caching, and challenge commonly held beliefs that MT cache tiers have to be pyramidal in size where fast, expensive but small tier-1 cache has to be backed up by a slow, cheap but large tier-2 cache device. We approach our second goal by using random forest as a model to predict the impact of adding a tier-2 cache on overall performance and by using sampling to reduce the overhead of MT cache analysis. The challenge is to understand the synergy between the workload and the storage devices on the server, which will ultimately determine whether MT cache is a good fit.

Turning the Storage Hierarchy On Its Head: The Strange World of Heterogeneous Tiered Caches Continuous growth in worldwide data storage has driven a commensurate increase in cache sizes and costs, creating the need for cost-efficient and workload-aware cache configurations. It is often more cost-efficient to employ multiple tiers of devices with varying cost-performance profiles than to rely on a single cache level. Conventional multi-tier caches are arranged hierarchically in a *pyramidal* organization, ranging from small, fast, expensive devices—to larger, slower, cheaper ones. Figure 1.1 shows an example of *pyramidal* and *non-pyramidal* MT caches. However, adding tiers is not always helpful; the benefit depends on *both the workload and the device properties.*

Exploring a large space of possible sizes through simulations can be computationally intractable. To facilitate efficient cache analysis, we introduce two novel metrics: a workload-based *Hit-Miss Ratio (HMR)* and a device-based *Overhead-Gain Ratio (OGR).* Together, they inform decisions on whether adding or resizing cache tiers would improve performance for a given workload. We present Cydonia, an HMR-based algorithm that finds cost-optimal tier sizes for a given workload, multi-tier cache configuration, and a cost limit, with minimal overhead.

We evaluated Cydonia with 136,476 points using 106 real-world workloads and device specifications for 17 diverse MT cache configurations and cost limits. On average, across all evaluations, Cydonia generates tier sizes exhibiting latencies that are higher than the cost-optimal ones identified with exhaustive search by 0.02–5.1% with mean of 2.3% (write-through policy) and 0.2–30% with mean of 13.1% (write-back policy)—while evaluating an average of only 0.2% (10 points) of the entire space of tier sizes.

Surprisingly, we also found that traditional pyramidal hierarchies are not always best; instead, a *non-pyramidal* configuration can have up to 33× lower latency compared to a single-tier cache and 25× lower latency compared to a pyramidal configuration of the same cost, highlighting the importance of proper tier sizing.

Large Scale Study of Multi-tier Caches Using Replay Cache simulations have shown that adding a tier-2 cache to an ST DRAM cache is an effective way to improve performance and cost-efficiency, but the improvement depends on workloads and device types. Some combination of devices can show significant improvement in performance, whereas others can harm performance even when the workload is constant. Simulations have also given insights regarding how the miss ratio is inadequate to size MT caches and that MT caches with *non-pyramidal* tier sizes—where a slower, smaller tier-2 cache follows a larger, faster tier 1—can sometimes perform better and at a lower cost compared to single-tier caches. However, simulations cannot model the complexity of an MT cache exactly, but only approximate it. Different SSD devices used as tier-2 cache react differently to different request rates, queue sizes, and access patterns, all of which cannot be accurately simulated. Cache engines that implement tier-2 caches are complex managing both performance and device lifetime [20]. The theoretical findings on MT caching must be corroborated with physical experiments.

We developed a general framework to investigate single- and multi-tier caches of varying sizes and properties. We built a tool to replay block traces and measure storage system performance, used it to analyze over 100 production block traces, and selected 13 representatives to replay on 3 different machine classes. In thousands of experiments totaling over 7.3 compute years, we studied various cache configurations. These extensive experiments show that adding a second tier helps or actually hurts performance. We found that adding an undersized tier-2 cache can *reduce* throughput by up to 35%. We also corroborated that *non-pyramidal* tier sizes can improve throughput compared to a single-tier cache. We identified workload characteristics and properties of devices in the storage system that favor MT caching over ST caching.

Although trace replay is accurate, it is also an expensive method of performance analysis. Because the space of possible configurations is so vast, we developed a decision tree model that accurately predicts whether adding a tier-2 cache of a given size would improve performance. Decision trees are transparent, which gives us insight into the features that are influential in determining when to add a tier-2 cache. Decision trees, with an accuracy of ∼90%, validate that workload properties other than miss ratio along with device properties influence the performance of systems with MT caches.

BlkSample: Sampling for Block Storage Traces Cache analysis is essential for cache optimization. Cache analysis techniques make a trade-off between overhead and accuracy. Working set analysis has minimal overhead but reveals little about the miss ratio. A workload with a smaller working set size does not necessarily have a lower miss ratio for a given cache size. Trace replay has high overhead but can accurately determine system throughput along with the miss ratio. Random spatial sampling has been able to get the best of both worlds, incurring low overhead while accurately approximating MRC of workloads. It can approximate the MRC with an average error of less than 0.01 while sampling less than 1% of the requests [96].

The miss ratio is an inadequate metric for MT cache analysis. The miss ratio does not translate linearly to performance metric such as throughput. Two workloads with identical miss ratio can have drastically different throughput. Other workload features such as request size, request rate, and write ratio along with performance of storage devices in the system influence the throughput. This is why random spatial sampling, which is oblivious to all workload features except for miss ratio, becomes inadequate to analyze the performance. We present, *BlkSample*, a sampling framework that combines the low overhead of random spatial sampling but also considers additional workload features that are influential in determining in MT cache performance. We find that random spatial sampling cannot represent features such as the request size even at a sampling rate as large as 80%. Random spatial sampling works with fixed-sized blocks, but block requests can be multiblock, accessing multiple blocks at once. Furthermore, random spatial sampling cannot generate a sample block trace but only generate an approximate MRC for a given workload. *BlkSample* supports multiblock requests and can generate block trace samples with a format identical to that of the source traces. We combine the reduced size of the sample and the accuracy of trace replay to see if we can estimate the throughput of the full workload based on the replay performance of the sample workload. The samples generated using *BlkSample* have a mean feature error of $\leq 10\%$ with a sampling rate of 10% compared to the mean feature accuracy of 40% using random spatial sampling when estimating the read/write request size, read/write interarrival time (IAT), read/write misalignment rate, and the write ratio while having a minimal effect on read/write MRC.

1.2 Thesis Overview

The remainder of the thesis is structured as follows. Chapter 2 provides the necessary background for the thesis. Chapter 3 establishes whether MT caches can improve on ST caches using the analysis of the miss ratio and the characteristics of various combination of storage devices that can be used to compose a block storage system. Chapter 4 explains the design and implementation of the block trace replay framework along with a large-scale study of replay experiments totaling ~ 7 years of runtime. The theoretical observations made in Chapter 3 are validated in Chapter 4 using physical experiments. Additionally, Chapter 4 takes the large corpus of replay data and trains decision tree models that predict whether adding a tier-2 cache of a given size improves performance given workload properties and the current tier-1 size. Chapter 5 introduces a sampling framework that preserves not only the MRC, but additional features that are integral to the performance of the MT cache, such as the mean read/write request sizes, the mean read/write IAT, the read/write misalignment rate, and the write ratio.

Chapter 2 Background

This chapter gives a background on block storage systems and MT caches. We discuss what separates block storage from other types of storage system and describe the peculiarities of block storage that have to be considered when evaluating them. We explore how MT architecture has been utilized in caching and elaborate on the multi-tier architecture that we use in this thesis.

2.1 Block Storage

2.1.1 Storage Architectures

The 3 primary types of storage systems are block storage, object storage, and file storage. Object storage stores data as objects of varying size in a flat structure. It can provide scalability and lower cost with limited complexity. However, objects cannot be partially edited; they have to be overwritten [82]. Continuously changing data, as in databases, is not suitable for object storage. It is useful to store static data such as video, audio, and log files. Content Delivery Networks such as Cloudflare, Akamai, and Binary Large Object Storage such as AWS S3, Azure Blob Storage are examples where object storage is used. File storage stores data as files in a hierarchical structure. File storage suffers from performance degradation as it scales. As more files, folders, and directories are added, the performance of searching and accessing information decreases. File storage is useful for sharing files and collaborating with multiple users. Block storage systems store data in fixed-size blocks and provide direct access to these blocks with low latency and high throughput. This makes block storage systems a good fit for distributed databases and high-performance computing, but its drawback is its high cost and complexity. In this thesis, we address the high complexity and cost of block storage by optimizing block storage servers using MT caches so that we need fewer block storage servers to meet performance demands.

2.1.2 Optimizing Block Storage

A large portion of the world's data is stored in block storage systems which are common in enterprise data centers and cloud computing [112] The Information Communication Technologies ecosystem is responsible for almost 10% of the world's energy consumption [71]. Cloud storage, including block storage, is a non-negligible part of the energy consumption. Lower costs and energy consumption are important to make block storage more accessible and reduce its load on the planet. A block storage system can span thousands of servers equipped with multiple storage devices. One way to improve the cost efficiency of block storage systems is to increase the performance of a single server. The throughput of a block storage server is largely dependent on the efficiency of its cache. The more data is served from the cache, the higher the throughput. MT caches increase the effective cache size and the potential throughput of block storage servers. Higher throughput from a block storage server means that fewer storage servers are required to meet performance requirements, which, in turn, results in lower cost and energy consumption.

2.1.3 Peculiarities of Block Storage

Block storage uses a cache for temporary high-performance storage and a backing store for persistence and capacity. The size of a block in the backing store can be different from the size of a block in the cache. For instance, the size of a sector in hard disk drives can be 512 bytes, whereas the size of a block in cache is typically 4KB. The fixed size of the cache block helps to efficiently utilize memory to store data. On the other hand, for object storage that handles a wide range of object sizes, special consideration has to be made to efficiently cache object of different sizes [20, 61].

The block trace we use in this thesis uses a sector size of 512 bytes and a cache block size of 4KB. Block requests are aligned with the sector size of the backing store, but not with the cache. Block requests that are misaligned with the cache require additional data to be loaded to the cache, as blocks in the cache are of fixed size. Misaligned read and write requests require additional reads to ensure that data are loaded into the cache at a fixed size. Misalignment can affect block storage performance as the additional read IO per misaligned writes can influence the write latency. Misalignment is specific to the block storage cache and not an issue in object caching where the entire object is cached or not.

2.2 Multi-Tier Cache

2.2.1 Multi-Tier: CPU vs. Storage

Multi-tier caching is not a novel idea. The use of multiple cache tiers, such as L1, L2, and L3 caches, in CPU architectures has been instrumental in reducing memory access latency and improving overall system performance [108]. However, the storage cache has traditionally been a single-tier DRAM. The use of multiple cache tiers in storage has become possible due to the large performance gap between traditional backing-store media, such as hard disk drives (HDDs) and modern solid-state drives (SSDs) [49, 48, 39] While we can translate insights from cache hierarchies in CPU caches, block storage caches face different constraints and challenges. The size of cache tiers in CPU cache is static and designed to support general workloads, whereas the size of cache tiers of block storage cache is dynamic and customized for each workload. We think of cache hierarchies in a pyramidal way because of the design of the CPU cache, where the lower tiers are slower and larger than the tiers above. We do not know if such design choices are optimal when sizing MT cache in block storage systems with a large space of possible tier sizes, including *non-pyramidal* where the second tier cache is smaller than the first tier cache. Furthermore, adding a second-tier cache can harm performance, and it is important to identify such scenarios.

2.2.2 Multi-tier Architecture

There are different types of MT cache architectures. One in which the RAM and the tier-2 cache are managed together [20]. The other is where the tier-2 cache is oblivious to the tier-1 cache and has no tuning control [15, 86]. We are looking at unified MT caches where both tiers of caches are managed by a central system, which in our case is CacheLib [20]. Numerous storage technologies (NetApp BlueXP [84], Oracle Exadata Database [3], Ceph [2], Redis [1]) have the capability to use flash as a cache device without clear workload and device guidelines. Research on using flash cache has largely focused on the SSD cache in isolation, ignoring the problems of sizing two tiers at once and identifying when not to add a second tier. The guides for flash cache largely focus on the reduction in miss ratio which is obviously the cache when we use flash to enlarge the cache but how much of an improvement in performance did adding a tier-2 cache bring is missing.

2.2.3 Multi-tier Analysis

Analysis of MT cache policies like Adaptive Replacement Caching (ReDARC) [23], Adaptive Level-Aware Caching Algorithm (ALACA) [23], and Adaptive Multi-level Cache (AMC) [22] focus on performance metrics like miss ratio, latency, and throughput, but ignore the cost-efficiency of MT cache. The miss ratios of tier-1 and tier-2 caches are translated to a single performance metric like IOPS based on the IOPS of the tier-1 and tier-2 cache device [56]. However, block caching has its own peculiarities, such as request types. Read and write misses have different effects on the system and overall performance.

Tier-2 Cache Device SSDs have asymmetric read/write performance, making the type of IO even more important [47]. The effect of device types on the effectiveness of MT caching has not been studied. Furthermore, there exist SSDs with diverse

Table 2.1: Workload Properties of 106 CloudPhysics traces.

Property	**Mean**	**Min**	**Max**	σ
Net I/O Size (GB)	101.0	0.9	1052.5	209.0
Read Size (GB)	95.0	0.2	983.0	201.2
Write Size (GB)	31.5	0.3	576.3	77.8
R/W	6.5	0.002	108.0	13.4

cost-performance profiles. Both the performance improvement and the cost efficiency of MT caching are dependent on the relative cost and performance of SSD to other devices in the system. The read/write performance of the tier-2 cache device under varying load determines the performance gain per read hit and the overhead from MT caching. The write amplification of SSDs also has to be managed when implementing a tier-2 cache [26].

Trace Replay Block I/O traces are a rich source of information on storage workloads. Workload features derived from traces can be used to infer the performance it would yield in a system. Workloads with majority sequential access pattern perform better than workloads with majority random access pattern. Workloads with low miss ratio perform better as most requests are served from the cache. An accurate measurement of performance cannot be made without details of the system. Even with system details, it is hard to accurately estimate system performance for millions of requests with different parameters.

Trace replay gives the most realistic estimate of how a system would perform for a given trace [37]. Replay can account for things like variation in device performance due to change in request rate and access patterns, which is very difficult to capture from workload analysis and simulation. The performance of SSDs is especially hard to estimate because the characteristics of SSDs can vary between types and vendors [66]. SSD can have drastically different read and write performance and is prone to latency spikes under high load. We use trace replay to obtain an accurate estimate of the performance of MT caches across various workloads and server types.

2.3 Workloads

For our evaluation, we selected the CloudPhysics traces [96]. There are 106 block I/O traces, each approximately one week long, collected from the disks of production VMware virtual machines in customer data centers running under the VMware ESXi hypervisor [93]. A user-mode application, deployed on each ESXi host, coordinated with the standard VMware `vscsiStats` utility [6] to collect complete block I/O traces from the virtual disks. These real-world traces cover a diverse set of characteristics showed in Table 2.1 that heavily influence caching behavior, making them suitable for evaluating MT caching. Many of these traces' properties have min-to-max ranges spanning an order of magnitude or more. For example, the traces have mean inter-arrival times ranging from 2.5ms to 250.0ms, total I/O request counts from 3.8M to 215.8M, mean I/O sizes from 4.0KB to 327.4KB, and write ratios from 0.033 to 0.996.

Chapter 3 Turning the Storage Hierarchy On Its Head: The Strange World of Heterogeneous Tiered Caches

3.1 Introduction

Modern storage systems host an enormous volume of data and must satisfy ever-increasing demands for faster response times. Caches are commonly placed in front of persistent storage to enable low-latency access, often via multiple devices arranged in tiers. Popular cache devices include NVMe SSDs, storage-class memory (SCM), and DRAM. Often, DRAM is backed with an additional cache tier to lower average storage latency or improve cost efficiency [32, 57, 75]. A storage administrator who wishes to configure the best caching configuration faces a complex landscape with diverse options for costs, capacities, and performance, even for a single device type.

The trivial performance-optimal configuration is always a large single tier with enough capacity to store the workload's entire working set. However, such configurations are usually cost-prohibitive, especially as working sets grow, so real-world systems often employ multiple tiers [28]. But it is challenging to find the optimal tier sizes for a multi-tier (MT) cache since available capital must be distributed across disparate devices with different sizes, costs, and performance.

For a given cost limit, workload, and available cache devices, the number of possible tier sizes can be vast, and different cache devices can be combined to form numerous MT cache configurations that must be evaluated. Figure 3.1 shows the numerous configurations that are possible at fixed cost. There are various configurations where the latency can be better or worse than the ST cache. MT cache analysis is necessary to identify cache configurations that lower latency and avoid ones that

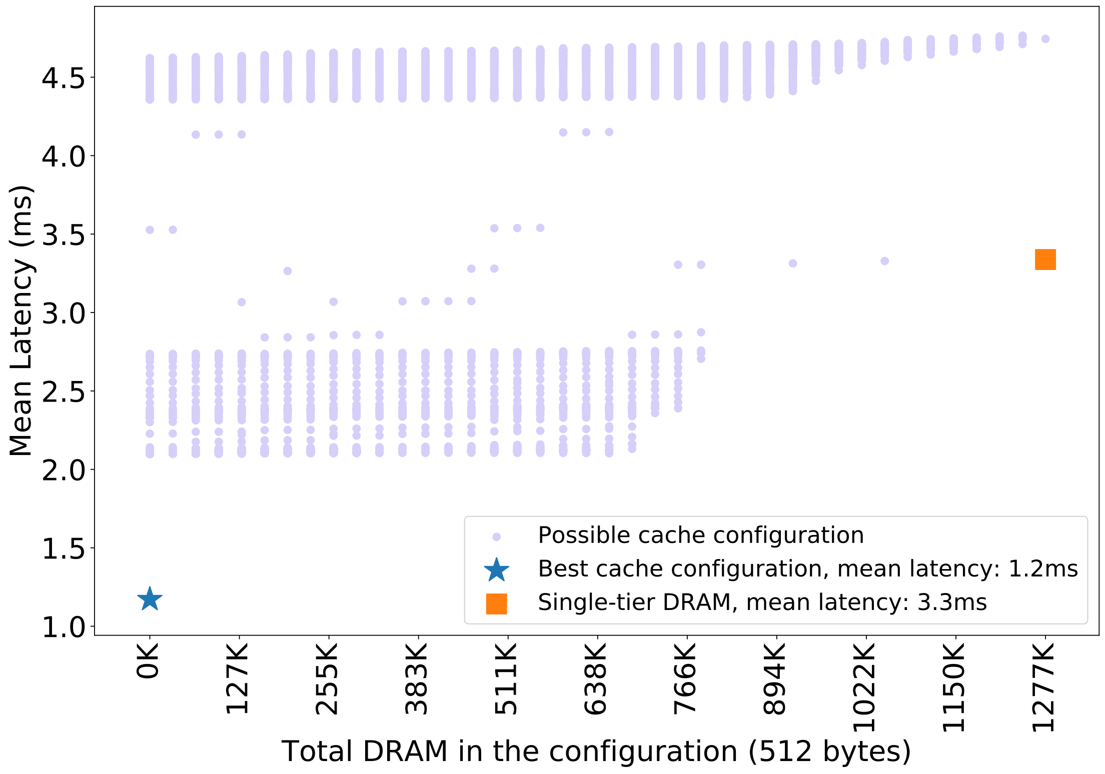


Figure 3.1: Multi-tier configurations (tier-1 size, tier-2 size) and mean latency for a fixed capital of \$500. We can see that there are various configuration that are worse and better than single-tier DRAM.

increase latency. In this work, our goal is to identify tier sizes that provide the *best performance for a given cost*, and to do so systematically for a given workload, multi-tier cache configuration, and cost limit, while evaluating a small portion of the space of tier sizes. This allows us to also evaluate a large number of possible multi-tier cache configurations given a device set.

A common approach to analyze single-tier cache performance is to compute miss ratio curves (MRC), which characterize miss rates for a given workload as a function of cache size and cache replacement policy [97, 79]. To model multiple cache tiers, eMRC [57] extended single-tier MRCs to associate cache misses with a tuple reflecting the size of each tier. However, this technique treats every cache miss the same, while the actual impact of cache misses varies in a MT configuration due to

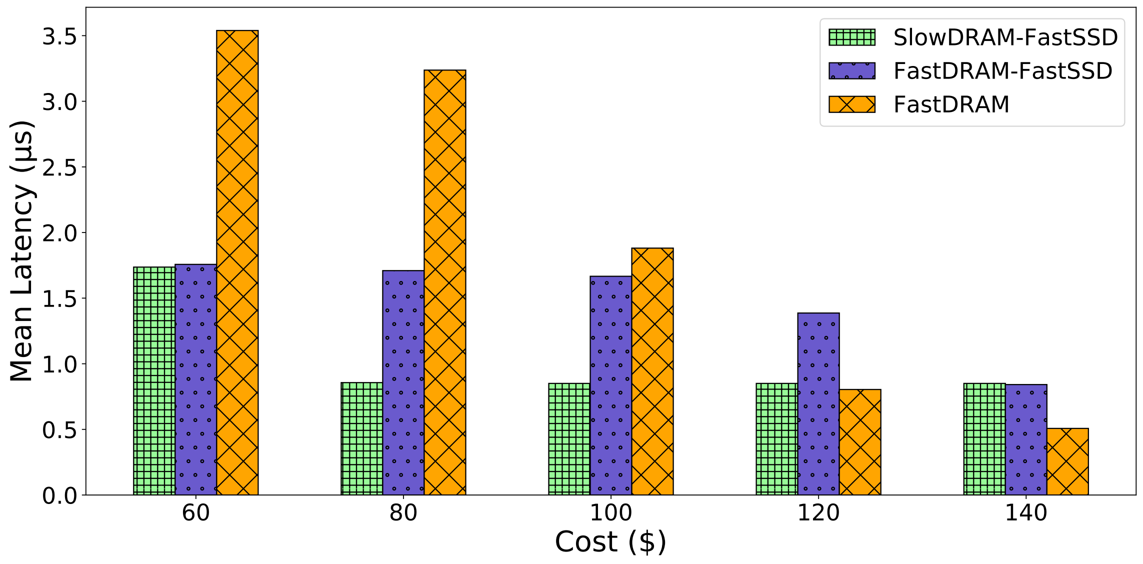


Figure 3.2: The relative mean latency of different combinations of cache devices can vary across cost limits for the same workload.

performance differences between devices. Furthermore, classical MRC models do not distinguish between read and write latency [109], despite the substantially asymmetric performance for these operations on devices such as flash memory.

Even in a system with two tiers, workload characteristics and device specifications need to be aligned to yield a cost-efficient cache. MT caching allows us to combine different devices to best fit the workload: DRAM can service requests with high locality, while an SSD reduces the number of requests that are served by slow back-end storage [42]. However,for different cost limits, the same workload can have different optimal multi-tier cache configurations. Figure 3.2 shows how the relative mean latency of 3 different combinations of cache devices changes along with the cost limit for the same workload.

We evaluated Cydonia using 17 different two-tier cache configurations on 106 real-world block I/O traces obtained from CloudPhysics [96]. We generate cost-effective tier sizes for each workload using two approaches: exhaustive search and Cydonia. The cost-optimal tier sizes obtained from the exhaustive search are used to measure the performance of the tier sizes generated by Cydonia. We analyzed the influence of capital on the optimal multi-tier configuration of a workload. We also examined these workloads to see if the traditional "pyramid scheme" of hierarchical caching—*i.e.*, that devices get slower and larger as they descend into tiers, holds true when a cost constraint is applied. We identified and analyzed several *counter-intuitive, non-pyramidal* configurations that, surprisingly, sometimes performed better than the traditional pyramid design. We also studied how a range of different capital constraints (*i.e.*, the cost-performance profile of the devices) affect the optimal cache configuration.

This chapter makes four key contributions:

1. We introduce two new metrics: a workload-based *Hit-Miss Ratio* and a device-properties-based *Overhead-Gain Ratio* to measure and predict cache performance.
2. We present Cydonia, a novel algorithm for determining low-latency and cost-effective tier sizes.
3. We analyze and present insights on the impact of capital and device properties on the optimal tier sizes.
4. We demonstrate that the conventional wisdom of using a pyramid scheme for tiered caching is not always optimal in the cost-aware MT cache scenario.

3.2 Design

There is no universal caching solution for even a single-tier cache; an optimal configuration for one workload can perform poorly for another. This workload-dependent cache performance is a major source of complexity. However, we found that the relationship between a workload and its optimal tier sizes can be better discovered using the HMR metric (Section 3.2.3). We leverage HMR to design Cydonia, an algorithm

Table 3.1: Device specifications and parameters. Prices are from amazon.com. Benchmarked specifications were collected from storagereview.com [13] and userbenchmark.com [88] in January 2022.

Label	Device	Type	Price	Capacity	Benchmarked Latency
FastDRAM	G.SKILL TridentZ Series	DRAM	\$120	16GB	0.0619μs read/write
SlowDRAM	G.Skill Ripjaws V	DRAM	\$68	16GB	0.0774μs read/write
FastSSD	Intel Optane SSD DC P4800X	SSD	\$1120	375GB	1.82μs read, 2μs write
MediumSSD	Intel DC S3700	SSD	\$454	800GB	13.33μs read, 27.77μs write
SlowSSD	Seagate IronWolf 110 NAS	SSD	\$132	480GB	18.18μs read, 33.33μs write
FastHDD	Seagate IronWolf Pro	HDD	\$644	20TB	120.8μs read, 974.6μs write
SlowHDD	Toshiba N300 NAS	HDD	\$289	8TB	1661.1μs read, 1037.3μs write

Table 3.2: 17 device combinations used for evaluation. Tier 1 and Tier 2 cache devices are combined with multiple backing storage devices.

Tier 1	Tier 2	Storage
FastDRAM	SlowDRAM	FastHDD, SlowHDD, SlowSSD
FastDRAM	FastSSD	FastHDD, SlowHDD, SlowSSD
FastDRAM	MediumSSD	FastHDD, SlowHDD
FastDRAM	SlowSSD	FastHDD, SlowHDD
SlowDRAM	FastSSD	FastHDD, SlowHDD, SlowSSD
SlowDRAM	MediumSSD	FastHDD, SlowHDD
SlowDRAM	SlowSSD	FastHDD, SlowHDD

that determines cost-efficient tier sizes for a given workload, device performance in terms of read/write latency, and cost limit—while evaluating significantly fewer points compared to exhaustive search.

3.2.1 Cache Allocation

We consider cache configurations containing devices with three representative types: DRAM, SSD, and HDD, using the detailed performance and cost specifications listed in Table 3.1. We use a 4KiB page size, a 1MiB unit allocation size, and 31B of metadata per page as in modern caching systems like CacheLib [21]. The metadata stores details such as the data key, size, and creation timestamp. This implies that, per unit of allocation, there are 1024KiB/4KiB=256 pages and the size of metadata per unit allocation is thus $256 \times 31 = 7,936$B (roughly two pages).

We calculate the cost of a configuration by including the cost of both data and metadata storage. Data can be stored in any tier, but we assume that the metadata is always stored in the tier-1 cache (DRAM). For example, given an MT configuration composed of FastDRAM (\$120 for 16GB) in tier 1 and FastSSD (\$1,120 for 375GB) in tier 2, the cost of a unit allocation (1MiB) of tier 2 is equal to the sum of the cost of cache space $\$1{,}120/(375 \times 10^3) = \$0.0023$ and the cost of metadata $(\$120/(16 \times 10^9)) \times 7,936 = \$0.00005952$, totaling \$0.00235952.

We assume that backing store space is infinite and do not consider its cost in our evaluations, since the size of the backing store is not relevant to cache design (and insufficient backing store is generally fatal for applications). When we compare the performance of multi-tier cache configurations, we evaluate a given workload and cost limit with a fixed backing store device. This cost model is representative of the flexibility available in a typical cloud configuration, where space can be allocated between different cache devices with relatively fine granularity. Our model can be readily modified to support other price points, device specifications, or allocation granularities.

3.2.2 MT Cache Configuration

We evaluated caches with two tiers, represented as a tuple of the tier-1, tier-2, and backing storage devices, such as (FastDRAM, FastSSD, SlowHDD). We consider only the sizes of tiers 1 and 2, since we assume backing store size is infinite. Note that even if a tier-2 cache is available, we allow its size to be 0 if including a tier-2 cache is not cost-optimal. Table 3.2 lists all the different tier-1, tier-2, and backing storage device combinations that we evaluated.

The latency of a tier is dependent on the latency of upper tiers. We use the average read and write latencies of cache devices in the tiers to compute the read and write latencies for each tier. To obtain the number of requests at each tier, we use the size of each tier to compute the sum of the corresponding reuse-distance histogram bins.

Once we have the latencies and the request counts for each tier, we can compute the average latency of the cache configuration for a workload.

3.2.3 Hit-Miss Ratio

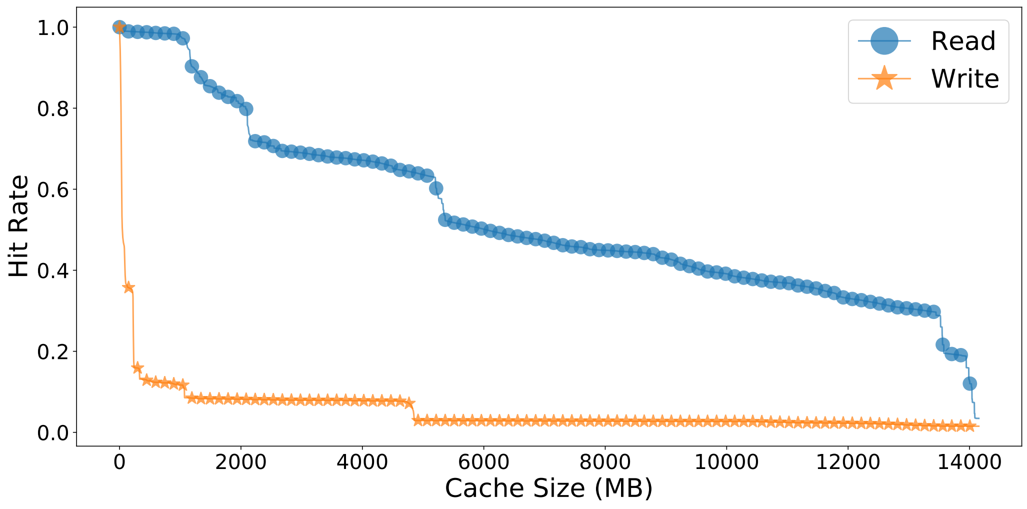
Consider deciding whether to add a second tier to a single-tier cache. Doing so is not trivial—adding a tier is not free, and an additional tier does not always translate to improved performance, despite having more aggregate cache space. In fact, an undersized cache tier can harm performance.

If the total cost of a caching system is kept constant, adding a tier will reduce the latency of some requests while increasing the latency of others. Whether the overall performance improves is not determined by the workload or the device properties alone. There is no workload that performs well on every cache, nor is there a caching solution that does well for every workload. The performance is determined by the interaction between the workload, the device, and the cache policies. The workload and the policy determine the number of hits and misses; the device properties control the latency reduction and overhead for hits and misses, respectively.

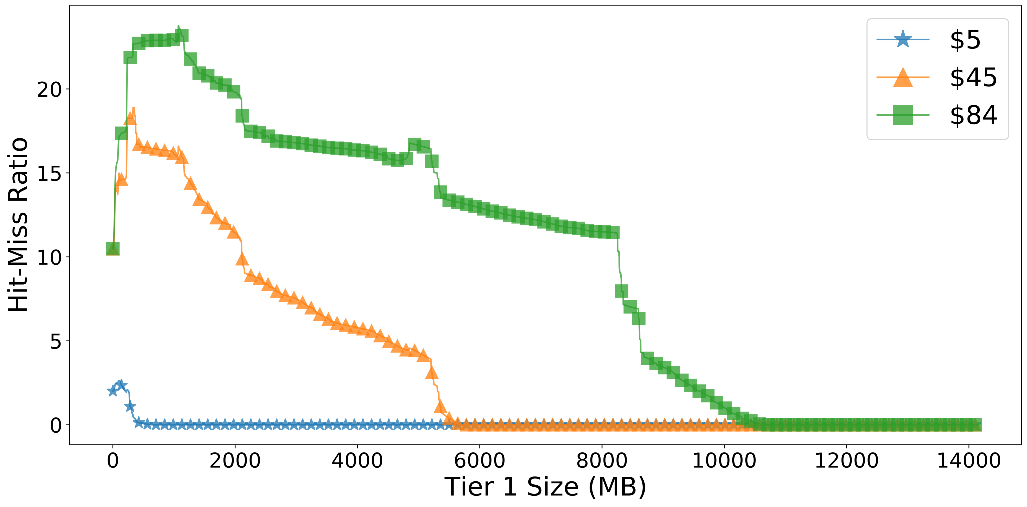
Consider two different caches, C_{ST} and C_{MT}, where C_{ST} is single-tier and C_{MT} is multi-tier. C_{ST} and C_{MT} use the same tier-1 device d_1, with read and write latencies (r_{d_1}, w_{d_1}), and backing store device d_s, with latencies (r_{d_s}, w_{d_s}). The read and write latencies for tier i of cache C are $(C(r_{t_i}), C(w_{t_i}))$. The latencies of a miss in C are $(C(r_{t_s}), C(w_{t_s}))$, since a full miss must go to the backing store.

The size of tier i of cache C is $C(s_{t_1})$. Since C_{ST} and C_{MT} use an identical tier-1 device, $C_{ST}(s_{t_1}) = C_{MT}(s_{t_1})$, $C_{ST}(r_{t_1}) = C_{MT}(r_{t_1})$, and $C_{ST}(w_{t_1}) = C_{MT}(w_{t_1})$. The difference is that C_{MT} has a tier-2 device d_2 with read and write latencies (r_{d_2}, w_{d_2}) and tier-2 latencies of $(C_1(r_{t_2}), C_2(w_{t_2}))$.

For a given workload W, we model the improvement from adding a second tier by comparing C_{MT} to C_{ST}. The read and write hit counts at tier i of cache C are



(a) Read/Write Miss Rate Curve (MRC)



(b) Hit-Miss Ratio Curve (HMRC) at different cost limits

Figure 3.3: Comparison of read/write Miss Rate Curve and Hit-Miss Ratio Curve of the same workload.

$(C(W_{r_{t_i}}), C(W_{w_{t_i}}))$; and the count of requests that miss all cache tiers is $(C(W_{r_{t_s}}), C(W_{w_{t_s}}))$, again because misses go directly to the backing store.

Latency reduction Adding a second tier of cache reduces misses. The tier-2 hits in C_{MT} would have been misses in C_{ST}, since the size of tier 1 is equal in both caches. The latency of a tier-2 read hit in C_{MT}, $C_{MT}(r_{t_2})$, is equal to the sum of read and write latencies of both tiers. The writes originate due to promotion of data from tier 2 to tier 1 and the corresponding eviction of some data back to tier 2:

$$C_{MT}(r_{t_2}) = r_{d_1} + w_{d_1} + r_{d_2} + w_{d_2}$$

(Note that we assume that the reads and writes involving the two devices are executed sequentially rather than in parallel.)

The latency of a read miss in C_{ST}, $C_{ST}(r_{t_s})$, is equal to the latency of a read from backing store and a write to the tier-1 cache:

$$C_{ST}(r_{t_s}) = r_{d_s} + w_{d_1}$$

We define the latency gain, G, per tier-2 read hit as the difference between $C_{ST}(r_{t_s})$ and $C_{MT}(r_{t_2})$:

$$G = C_{ST}(r_{t_s}) - C_{MT}(r_{t_2}) = r_{d_s} - r_{d_1} - r_{d_2} - w_{d_2}$$

Note that a tier-2 write hit does not yield any latency reduction but instead incurs additional overhead. We discuss this case next.

Latency overhead Adding a second tier introduces latency to certain requests. The latencies of a tier-2 write hit $(C_{MT}(w_{t_2}))$ and write miss $(C_{MT}(w_{t_s}))$ in C_{MT} are the same and are equal to the sum of read and write latencies of tier 1, write latency of tier 2, and backing store:

$$C_{MT}(w_{t_s}) = C_{MT}(w_{t_2}) = w_{d_s} + r_{d_1} + w_{d_1} + w_{d_2}$$

The tier-2 write hits and write misses in C_{MT} would have been write misses in C_{ST}. The latency of a write miss in C_{ST}, $C_{ST}(r_{t_s})$, is equal to the latency of a write to backing store and a write to tier 1:

$$C_{ST}(w_{t_s}) = w_{d_s} + w_{d_1}$$

We define the overhead, O_w, per tier-2 write hit and write miss (tier-1 write miss) as the difference between $C_{MT}(w_{t_2})$ and $C_{ST}(w_{t_s})$:

$$O_w = C_{MT}(w_{t_2}) - C_{ST}(w_{t_s}) = r_{d_1} + w_{d_2}$$

We now compare read misses in C_{ST} and C_{MT}. The latency of a read miss in C_{ST}, $C_{ST}(r_{t_s})$, is equal to the latency of a read from backing store and a write to tier 1:

$$C_{ST}(r_{t_s}) = r_{d_s} + w_{d_1}$$

The latency of a read miss in C_{MT}, $C_{MT}(r_{t_s})$, is equal to the sum of the read and write latencies of the tier-1 device, a write to tier 2, and a read from the backing store:

$$C_{MT}(r_{t_s}) = r_{d_s} + w_{d_1} + r_{d_1} + w_{d_2}$$

We define the overhead O_r per read miss as the difference between $C_{MT}(r_{t_s})$ and $C_{ST}(r_{t_s})$:

$$O_r = C_{MT}(r_{t_s}) - C_{ST}(r_{t_s}) = r_{d_1} + w_{d_2}$$

Observing that the overheads for a tier-2 write hit, write miss O_w, and read miss O_r are all equal, we denote it simply as overhead O:

$$O = O_r = O_w$$

Hit-Miss Ratio The definition of a cache hit is understood to mean that a block is found in the cache, but it does not differentiate between read and write requests. For an MT cache, we define a *cache hit* as a request whose latency is decreased and a *cache miss* as a request whose latency is increased. A cache hit H, when a tier-2 cache is added, refers to tier-2 read hits, and a cache miss M refers to tier-2 write hits, write misses, and read misses. When computing HMRC based on a cost-limit C, as shown in Figure 3.3b, for each tier-1 size of cost C_{T1}, the corresponding tier-2 size and the tier-2 read/write hits are derived based on the remaining cost $(C - C_{T1})$.

The total latency reduction is $H \times G$ and the total latency increase is $M \times O$. For latency reduction to offset the latency increase, we have:

$$G \times H > O \times M \quad \Longrightarrow \quad \frac{H}{M} > \frac{O}{G}$$

In the above inequality, we have isolated the device and workload properties into two separate metrics: the Hit-Miss Ratio (left) and the Overhead-Gain Ratio (right). Together, they can be used to determine whether a given tier-2 cache can improve performance for a storage system with a fixed backing storage device and a single cache tier. If the Hit-Miss Ratio at a given tier-1 size is lower than the Overhead-Gain Ratio, then an additional tier will not improve performance. This insight allows us to reduce the analysis space to find cost-optimal tier sizes based on the device properties. (Note that these values would be different for a cache using a different admission policy (*e.g.*, inclusive) and/or replacement policy (*e.g.*, ARC), but a similar trade-off analysis still holds.)

3.2.4 Cost-Efficient Cache Algorithm

Hit-Miss Ratio Curve (HMRC) is generated from a reuse distance and a cost limit as discussed in Section 3.2.3. The `hmrc` function in Algorithm 1 refers to this process. It is not represented in a separate algorithm for brevity. The HMRC is then used in Algorithm 1 to generate cost-efficient tier sizes given a reuse-distance histogram, cache devices at each tier, and a cost limit, evaluating fewer points than an exhaustive search.

In Algorithm 1 we first establish a baseline by evaluating the mean latency of an ST cache sized to meet a cost limit, as shown in Line 9. The `eval` function refers to the evaluation of the mean latency of a cache, given the tier sizes, MT cache devices, and reuse distance histogram. This can done by either simulation or analysis-based evaluation. The remainder of Algorithm 1 searches for a combination of tier-1 and tier-2 sizes that offer a mean latency lower than an ST cache of the same cost. In Line 10 we create an array of indexes `i` into `HMRC`, sorted by the value of `HMRC[i]`, so that $\texttt{HMRC}_{\texttt{sort}}[0]$ will be the index of the entry in `HMRC` that has the highest hit-miss ratio. Note that here, `i` also reflects that the size of the tier-1 cache and tier-1 cache sizes with hit-miss ratios lower than the `OGratio` are filtered out as discussed in Section 3.2.3. We evaluate the tier-1 sizes in order of Hit-Miss Ratios (HMR), based on the assumption that sizes with higher HMRs are more likely to outperform the ST baseline. Thus, the tier-1 size with the maximum HMR and its corresponding tier-2 size, derived from the cost limit, is the first MT point evaluated in line 15; we store that size in `T1`.

Based on the result of the first point evaluated, we pick the tier-1 sizes to evaluate. If the first MT point performs *better* than the ST baseline, we search only smaller tier-1 sizes, discarding the larger ones (Line 20). Similarly, if that first MT point performs worse than the ST baseline, we choose to search only tier-1 sizes larger than `T1` (Line 25).

3.3 Evaluation

We used real-world workloads and device characteristics to analyze MT caches. Our goal was to validate and rethink assumptions about cache configuration, and to optimize as we move from ST to MT caches. We used exhaustive search to evaluate

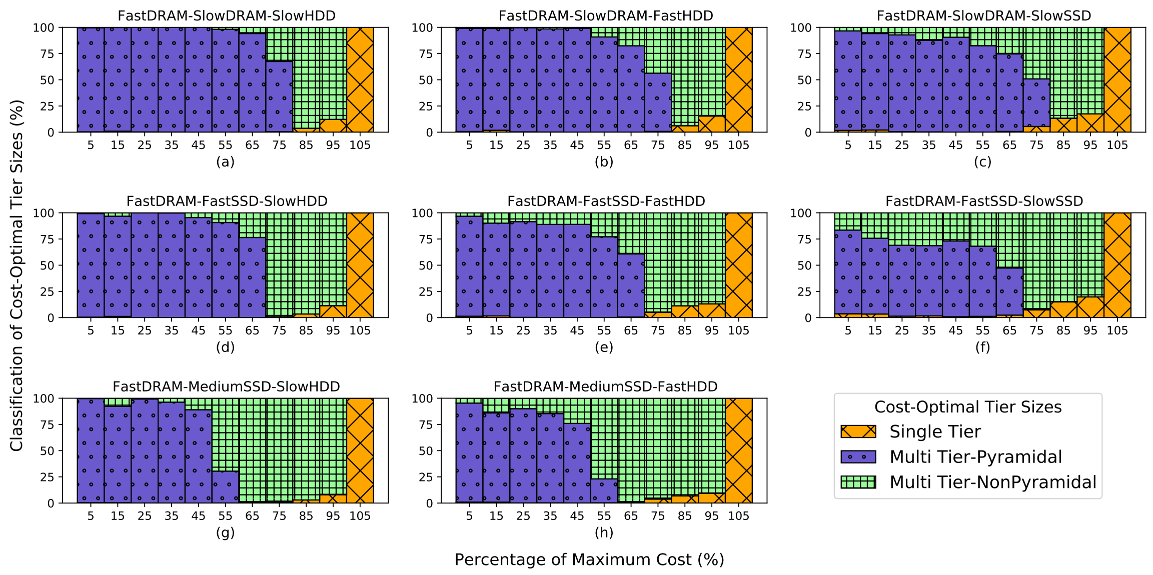


Figure 3.4: Each plot shows the percentage split between optimal tier sizes at various scaled cost value for an MT cache configuration. Scaled cost represents the cost value as a percentage of the maximum cost. Maximum cost is the value at which the working set size fits in the most expensive, fast cache device, which in this case is FastDRAM.

cost-optimal MT cache configurations from 106 workloads, 17 device combinations, and cost values scaled based on the size of the working set size. We found counter-intuitive non-pyramidal MT caches to be cost-optimal; we modeled the device combinations and cost constraints under which such configurations are preferable, as well as traditional pyramidal MT caches and ST caches. We used the cost-optimal cache identified from exhaustive search as a baseline to measure the accuracy of our HMR-based algorithm, which generates cost-efficient cache configurations while evaluating fewer points.

3.3.1 Cost-Optimal Tier Sizes

As shown in Table 3.1, we used two types of memory devices, *FastDRAM* and *SlowDRAM*, as tier-1 cache, along with a variety of tier-2 and backing storage devices. These resulted in diverse device combinations for MT caches, as illustrated in Table 3.2. The cost-optimal tier sizes for all 106 workloads, 17 MT cache configurations, and cost limits were generated through exhaustive search; the cost-optimal types were determined by relative sizes of tier 1 and tier 2 in the cost-optimal tier sizes. Figure 3.4 shows the classification of cost-optimal tier sizes—ST (single-tier), MT-P (pyramidal), and MT-NP (non-pyramidal)—across different cost values, represented as a percentage of the maximum cost. In each subfigure row in Figure 3.4, going from left to right, the tier-1 and tier-2 cache devices are constant while the backing storage device gets faster. Going from top to bottom in each column of subfigures, the tier-1 and backing-storage devices are constant while the tier-2 device gets slower. MT tier sizes (pyramidal shaded as dots and non-pyramidal shaded as squares) are cost-optimal in the majority of instances across cost limits and MT cache configurations.

The consistent pattern in all plots is that non-pyramidal tier sizes are prevalent at higher cost values. This is because at high costs we can use a cheap tier 2 to hold the workload's entire working set in the cache while still having a large tier-1 size. As the backing storage gets faster while the tier-1 and tier-2 cache devices remain constant, we see an increase in instances where single-tier and non-pyramidal configurations are optimal, and a corresponding decrease in instances where pyramidal configurations are optimal. When the storage device is slow, cache misses incur high latency. To minimize cache misses and consequent access to slow backing store, tier sizes with a large tier-2 cache size become cost-optimal. As the backing store gets faster, cache misses incur less latency, favoring a smaller tier 2 or not having a tier 2 at all, which in turn benefits single-tier and non-pyramidal configurations.

As the tier-2 device gets slower and cheaper while holding the tier-1 and backing storage devices constant, we see a decrease in instances where ST and pyramidal configurations are optimal and an increase in instances where non-pyramidal configurations are optimal. Compared to Figure 3.4(d), Figure 3.4(g) shows that non-pyramidal configurations become optimal at lower cost limits. This is because the tier-2 device used in Figure 3.4(g) is cheaper and we can thus fit the entire working

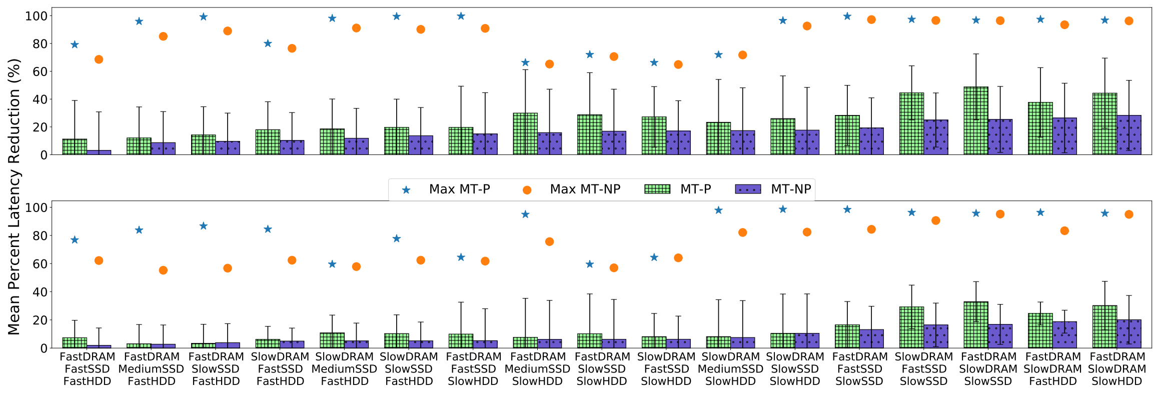


Figure 3.5: Mean percent latency reduction from *pyramidal* (MT-P) and *non-pyramidal* (MT-NP) MT caches, compared to an ST cache of the same cost limit, across 17 MT cache configurations. The data is sorted in ascending order of the mean percent latency reduction from non-pyramidal MT caches. The upper and lower rows represent write-back and write-through policies, respectively.

set in the cache for a lower cost, giving rise to non-pyramidal configurations at lower cost values.

Figure 3.4 shows the fraction of cost-optimal non-pyramidal tier sizes. Figure 3.5 shows the mean and maximum percentage improvement of pyramidal and non-pyramidal configurations over an ST one, given the same cost limit across all workloads, for write-back and write-through policies for each of the 17 MT cache configurations. The error bars represent standard deviations. The latency improvement due to the write-through policy is smaller than that for write-back. This is because write-through caches write to the backing store on every write request, incurring high latency that cannot be avoided by any cache. However, the potential for latency reduction from tier-2 read hits is still the same for both write-back and write-through caches.

Figure 3.5 is sorted by ascending mean percent latency reduction from *non-pyramidal* MT caches. We can see that the top five rightmost MT caches give two key insights. The first is that the MT cache configurations using FastDRAM and SlowDRAM as tier-1 and tier-2 caches have the top three highest-performing non-pyramidal tier sizes on average, with mean percent latency reductions of 25%, 26%, and 28%, respectively. A fast tier-2 cache device means high latency gain per tier-2 read hit and low overhead on a tier-2 write miss. This means that a smaller tier-2 cache with a lower hit ratio can be sufficient to improve performance over ST configurations.

The second insight is that the top five rightmost MT caches in Figure 3.5 contain all three MT cache configurations using SlowSSD as the backing storage device, which is the fastest storage device we evaluated. This further confirms the pattern we saw in Figure 3.4, where improved performance of the backing storage device causes non-pyramidal multi-tier caches to become more prominent.

3.3.2 Hit-Miss Ratio (HMR) Algorithm

Cydonia evaluates a selection of points, guided by the HMR, to come up with cost-efficient MT tier sizes for a given workload, MT cache configuration, and cost limit. We evaluated Cydonia on the same 106 workloads and 17 MT cache configurations, again for both write-back and write-through policies. We compared the algorithm's performance across all types of MT cache configurations while evaluating the first ten tier sizes identified by Cydonia. We chose the first ten sizes because the performance improvement per additional evaluation after the first ten becomes small. We next evaluated the following aspects of Cydonia:

- How often does Cydonia identify MT tier sizes that improve upon an ST tier size of the same cost limit, even if the identified MT tier sizes are not the exact cost-optimal sizes found through exhaustive search?
- What is the potential latency reduction in MT tier sizes that were successfully identified by Cydonia—and in those that were not?
- What is the difference in latency between the MT tier sizes identified by Cydonia and the cost-optimal sizes found using exhaustive search?

Figure 3.6 displays the percent difference between the latency of MT tiers whose sizes were identified by Cydonia and the cost-optimal MT tier sizes identified by exhaustive search. The bars are in ascending order of mean percent latency difference of write-back caches; the error bars represent the standard deviations. The mean percent latency difference ranges from 0.2% to 30.7% for write-back caches and 0.02% to 5.3% for write-through caches. The top five MT cache configurations with the highest mean percent latency difference fall into three categories. The first is where the cost-performance profiles of tier-1 and tier-2 cache devices are close: FastDRAM and SlowDRAM. The second is where the backing storage device is fast: SlowSSD. The third is where the backing storage device is slow: SlowHDD. These categories apply to both write-back and write-through MT cache configurations. The top five MT cache configurations with the lowest write-back mean percent latency difference, <5% for write-back and <0.35% for write-through, use the backing storage device FastHDD.

Depending on the multi-tier cache configurations, in 21–85% of the time, Cydonia fails to find any MT tier sizes that improve upon the ST size, when such tier sizes exist. However, the mean latency reduction potentials when MT tier sizes are identified in write-back and write-through caches are 46% and 21%, respectively. On the other hand, the mean latency reduction potential when MT tier sizes are not identified in write-back and write-through caches are 7% and 2%, respectively.

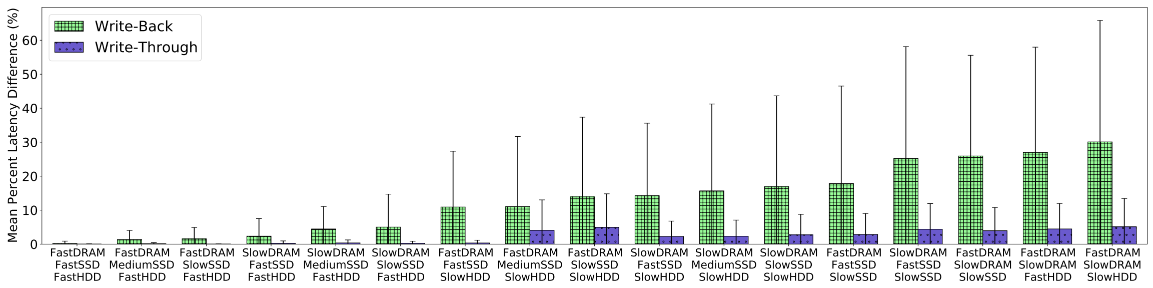


Figure 3.6: The mean percentage difference between the latency of the MT tier sizes identified by Cydonia and the cost-optimal MT tier sizes. The bars are sorted in ascending order of mean percent latency error of write-back caches.

3.4 Related Work

Performance metrics such as raw throughput, latency, and hit/miss ratio have been used to evaluate multi-tier caching policies such as Adaptive Replacement Caching (ReDARC) [23], Adaptive Level-Aware Caching Algorithm (ALACA) [23], and Adaptive Multi-level Cache (AMC) [22], but all of these disregard the capital expenditure associated with engineering a multi-tier cache. Existing multi-tier point solutions for virtual machines [107, 106, 62, 75], cloud storage [24], and optimized file systems [46] lack generality and focus on limited configuration spaces or specific environments. Our approach is general and can be applied to any MT cache configuration and workload.

Miss-ratio curves (MRCs), equivalent to hit-ratio curves (HRCs), are an effective way to approximate the working set size of a particular workload, and can be efficiently constructed for caches with stack-based replacement policies by using reuse distances [60]. UMONs [74], Fractals [38], CounterStacks [99], and PARDA [64] employ reuse distance access counters to keep track of stack distance in each cache size, the total number of accesses and the unique number of accesses in a workload. Sampling techniques have been developed to reduce the space complexity associated with reuse distance analysis, such as AET [40] and SHARDS [96]. Recent work by Zhang *et al.* [109] considered two-tier hierarchies with asymmetric read/write performance and also extended the spatial sampling results on HRCs to enable fast simulations. Fu *et al.* [35] have shown that traditional MRC analysis is often inadequate for cache partitioning, since all requests are treated equally. However, these efforts do not provide an alternative metric, such as the hit-miss ratio, that can aid in sizing tiers.

3.5 Conclusion

As workload working sets continue to increase and new cache devices are developed, experimentation with new cache architectures, with an eye toward cost, is essential for provisioning modern storage systems. Hit-Miss ratios aid in MT analysis by illustrating the potential of adding a new tier of cache, which miss-based or hit-based metrics fail to do. Using Hit-Miss ratios, Cydonia finds low-latency tier sizes based on workload, cost limit and MT cache configuration, while evaluating as little as 0. 2% of the entire search space of tier sizes, with a mean percentage latency difference of 13.1% and 2.3% for write-back and write-through policies, respectively. Our results highlight the viability of MT caches with quick reconfigurability and enable storage administrators to quickly evaluate a large number of MT cache configurations and choose the best fit for the workload.

Our analysis also reveals that the traditional hierarchical approach, where the device gets larger as we go down the tiers, can be sub-optimal for some costs and workloads. Especially when the budget is high, non-pyramidal tier sizes can be optimal 85% of the time on average. As the tier-2 cache device and backing store get faster and cost limits increase due to demanding performance requirements, the usefulness of non-pyramidal tier sizes also increases. Therefore, multi-tier cache design should consider the larger space of possible configurations, making hit-miss ratio curves and an algorithm such as Cydonia critical to multi-tier cache design.

```
Result: Size array
// Inputs:
rdhist ← the reuse distance histogram;
C ← cost limit ;
MTcache ← list of devices at each tier ;
step ← step-size ;
$eval_{max}$ ← maximum number of evaluations ;
// Initialization:
OGratio ← overhead-gain ratio of MTcache ;
HMRC ← hmrc (rdhist, C);
$T1_{MAX}$ ← size of tier 1 of MTcache costing C ;
$LAT_{ST}$ ← eval($T1_{MAX}$, 0, MTcache, rdhist);
$HMRC_{sort}$ ← array of indices into HMRC, sorted by corresponding HMRC value where
  HMRC ¿ OGratio ;
T1 ← $HMRC_{sort}$[0] ;
$C_{T1}$ ← cost of T1 ;
$C_{T2}$ ← cost of C − $C_{T1}$ ;
T2 ← max T2 size of cost $C_{T2}$ ;
// Evaluate highest hit/miss ratio
LAT ← eval(T1, T2, MTcache, rdhist);
// Decide whether to search larger or smaller
// tier-1 sizes
if LAT < $LAT_{ST}$ then
    $OPT_{T1}$ ← T1 ;
    $OPT_{T2}$ ← T2 ;
    $LAT_{MIN}$ ← LAT ;
    $T1_{LIST}$ ← $HMRC_{sort}$ where $HMRC_{sort}$ ¡ T1;
else
    $OPT_{T1}$ ← $T1_{MAX}$ ;
    $OPT_{T2}$ ← 0 ;
    $LAT_{MIN}$ ← $LAT_{ST}$ ;
    $T1_{LIST}$ ← $HMRC_{sort}$ where $HMRC_{sort}$ ¿ T1;
// Evaluate sizes in order of hit/miss ratio
$EVAL_{COUNT}$ ← 1;
for $T1_{index}$ ← 0 to len($T1_{LIST}$), $T1_{index}$+ = step do
    T1 ← $T1_{LIST}$[$T1_{index}$] ;
    $C_{T1}$ ← cost of T1 ;
    $C_{T2}$ ← C - $C_{T1}$ ;
    T2 ← size of tier 2 of MTcache costing $C_{T2}$;
    LAT ← eval(T1, T2, MTcache, rdhist);
    if LAT < $LAT_{MIN}$ then
        $OPT_{T1}$ ← T1 ;
        $OPT_{T2}$ ← T2 ;
        $LAT_{MIN}$ ← LAT ;
    $EVAL_{COUNT}$ ← $EVAL_{COUNT}$ + 1;
    if $EVAL_{COUNT}$ == $EVAL_{MAX}$ then
        break ;
return $OPT_{T1}$, $OPT_{T2}$, $LAT_{MIN}$ ;
```

Algorithm 1: Algorithm to generate cost-efficient tier sizes given a workload, MT cache configuration, and cost limit based on user-defined step size and evaluation limit.

Chapter 4 Large Scale Study of MT Caching Using Trace Replay

4.1 Introduction

To support fast response times for workloads with large working set sizes, block storage systems require large caches. The maximum throughput for a given workload and a storage server is obtained when the DRAM fits all the blocks accessed in the workload [29]. DRAM is limited in size and expensive, using a DRAM large enough to fit the working set size is rarely possible and is likely wasteful. A DRAM cache, which we will call *single-tier* (ST), can potentially be extended with an additional SSD tier to form a multi-tier (MT) cache to increase throughput while reducing performance per dollar cost, the DRAM footprint [33, 56, 76] and potentially the energy footprint.

A plethora of storage devices with varying performance and cost characteristics can be used as tier-2 caches and backing stores. The choice of tier-2 and backing store devices influences how throughput is impacted when a tier-2 cache is added. But the tier-2 cache must be appropriately sized; in fact, we have found that adding an undersized tier-2 cache can lead to a hierarchy that has lower throughput compared to not having a tier-2 cache at all.

A common approach to analyzing single-tier cache performance is to construct Miss Ratio Curves (MRCs), which characterize miss ratios for a given workload as a function of cache size and replacement policy [97, 79]. To model multiple cache tiers, eMRC [57] extended single-tier MRCs to calculate miss-ratio surfaces, and employed simple models of cache throughput that assume performance improves linearly with hit ratio.

In Chapter 3, we evaluated 106 block storage workloads with 16 combinations of tier-1 cache, tier-2 cache, and backing store to evaluate the effectiveness of MT caching. The miss ratio at each tier, the vendor reported performance of each device along with its retail cost was used to determine that MT caches can improve performance and cost efficiency of block storage servers. We identified that a large performance gap between cache tiers and backing store along with workloads with high tier-1 miss ratio favored MT caching. We found that non-pyramidal tier sizes, where a fast, expensive tier-1 cache can be followed by a smaller, slow, but cheap tier-2 cache can improve performance and cost efficiency of the storage server. However, these findings are based on simulation and analysis, which are fast but make some assumptions that make them more prone to error than physical experiments. We assume that device performance is static and largely ignore the role request rate plays in latency of cache tiers and backing store.

To better understand the effect of the request rate on the performance of MT caches and validate our finding from Chapter 3, we implemented a block-trace replay framework that measures the performance of the various components of the block storage system. We ran thousands of experiments on physical hardware, which amounts to several years of compute time, while varying the cache configurations, machines, and workloads. In this chapter, we present the results of those experiments, including analyses of the overhead of adding a tier-2 cache, the effect of request rate on MT cache performance, the size of tier-2 cache required to improve performance, and of the cost implications of such configurations. It is not trivial to predict whether adding a tier will help, because it depends on the interaction of multiple devices that have their own parameters and complexities. Therefore, we used data from block-replay experiments to train a decision tree (DT) machine learning model to make this prediction (see Section 4.3.9 for details). Given block workload features and proposed sizes for both tiers, the model predicts whether the configuration with these sizes would outperform a single-tier cache with the same tier-1 size.

When evaluated under varying performance improvement thresholds, the decision trees for machines c220g1 and c220g5 achieved mean accuracies of 88% and 90%, precisions of 88% and 89%, and recalls of 88% and 87%, respectively. Decision trees support quick inference using simple features generated from the trace; they are also explainable and interpretable, which helps a storage designer understand the conditions under which adding a given tier-2 cache will improve performance.

We explored the following questions to understand when to use a tier-2 cache: (1) What combination of devices favors multi-tier caching? What tier sizes are performance-efficient? (2) Are multi-tier caches dollar-cost efficient? (3) How do we pick tier sizes? How do those sizes influence the performance and cost of a multi-tier cache? (4) Do tier sizes need to be pyramidal, where larger and slower cache tiers follow a smaller and faster tier? Our results reveal the ways in which the performance differences between storage devices in a given machine and the selection of tier sizes influence the cost and performance of multi-tier caches.

This chapter makes three key research contributions:

1. We implemented a framework that measures the performance of different components for both single-tier and multi-tier caches. We replayed 13 representative production block traces on three classes of bare-metal cloud machines, consuming a total of 7.3 compute years. We have derived insights from the data collected regarding the types of machines and tier sizes to use to maximize the benefits from MT caching and avoid poor configurations, such as when adding a second tier *hurts* performance.

2. Our experiments validated that the conventional wisdom of using a pyramidal scheme for tiered caching is not always the best way to trade off performance and cost; for example, if the cost of DRAM is 6× that of tier 2 cache, then over 90% of MT caches with non-pyramidal configurations were more cost-effective

than ST caches with larger tier-1 sizes.

3. We have trained decision-tree models for machine c220g1 and c220g5 that can infer whether adding a tier-2 cache to an ST cache will improve performance, for a given workload and tier-1 size, with average weighted precision of 88% and 89%, recall of 88% and 87% across different performance improvement thresholds.

4.2 Design and Implementation

To study multi-tier cache performance, we designed and built a trace replay and measurement framework to study the influence of workload, machine, and cache configurations on cache performance. Trace replay is the most accurate method to assess the performance of a block storage system for a given workload. We implement a replay framework for block traces that tracks live statistics during trace replay and aggregate statistics once the replay is completed. We replayed 13 representative production workloads on 3 classes of bare-metal cloud machines while varying the cache configurations. Statistics collected during replay include measurements of performance of memory allocation, find function, tier-2 read/write, backing store read/write, and more. The recorded performance along with the workload properties will help us better understand the impact of adding a tier-2 cache on the performance of a block storage system. Using workload features and performance metrics collected during replay, we trained a decision tree that predicts whether adding a cache tier will improve performance. In this section, we will start with a brief overview of the design followed by a description of each component of the replay framework.

4.2.1 Overview

Figure 4.1 shows that the replay framework is made up of three main components: replay application, cache, and storage. The replay application is responsible

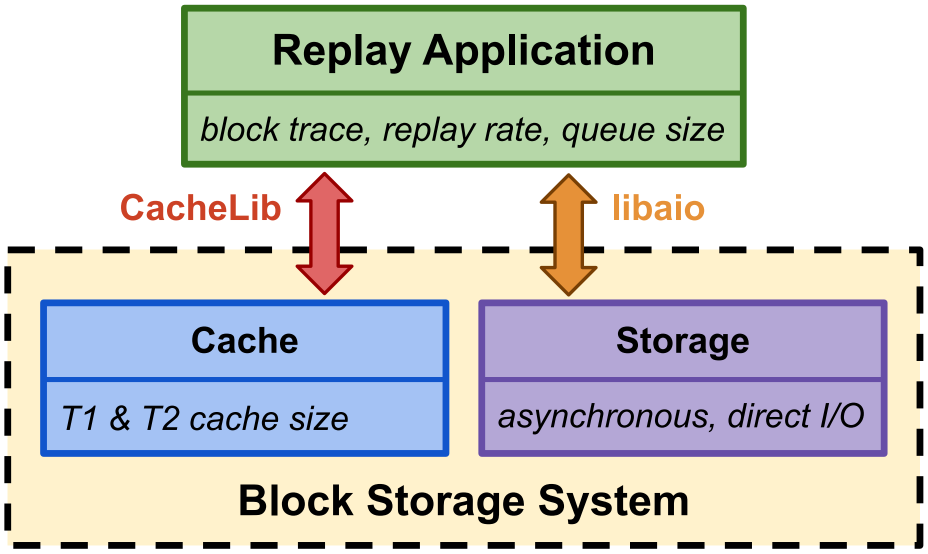


Figure 4.1: Block-trace replay framework design.

for tracking a block request from its creation, when its read from the block trace, to its conclusion, when all the required bytes are processed and the statistics updated. To successfully process all bytes requested in a block request, the replay application communicates with the cache and storage as required. It uses CacheLib, an open-source cache engine, and libaio to communicate with the two main components of a block storage system, cache and storage, respectively. The block cache can be DRAM-only (ST) or DRAM and SSD (MT) and the backing store can be HDD or SSD. The following sections will describe each component of the replay framework in detail.

4.2.2 Replay Application

The goal of the replay application is to stress the block storage system according to the block trace and to track the necessary statistics. To track temporal statistics such as latency, we used the total cycle count of the CPU during the lifetime of the block request. On modern CPUs, the instructions ”rdtsc” and ”rdtscp” can be used to track the cycle count between two points in the code [4]. The cycle count recorded from the CPU will have a higher resolution than the traditional method of getting time from the OS. The cache operates in block-sized units; for our experiments, we used 4KB cache blocks. A pending request queue tracks all the block requests being processed in the system. Each block request is processed asynchronously, and the number of requests that can be processed at the same time is equal to the size of the pending queue. To fulfill its function, it relies on three different types of threads: replay, block request processor, and backing request processor.

Replay A single replay thread is used to read block requests from the trace and add them to the queue of pending requests. Using a single thread ensures that the order of submission of requests to the block storage system is the same as the block trace. Once the block request is read from the trace, there is a waiting time equal to the inter arrival time (IAT) before adding the block request to the pending queue. Replaying block requests with a low IAT can be impossible if the IAT value is less than the time it takes to read a block request from the trace. To avoid this, we read block requests in batches if the requests have a low IAT. The framework also supports stressing the system by replaying traces faster than their IAT. There are two ways to accelerate trace replay and not follow the IAT of the trace for all block requests. The first is that if the application queue is empty, we ignore the IAT and issue the next request immediately; doing so allows us to skip idle times in the trace. The second method is to use a user-defined *replay rate*, which divides all IATs of block requests by this value. Thus, a replay rate of 1×, used for most of our experiments, means that the trace will be replayed with its original IATs, while a rate of 2× doubles the replay speed by halving the IATs. If the queue of pending block requests is full, the application waits for an empty slot before issuing the next request; no requests are dropped.

Block Request Processor Processor threads read requests from the pending block request queue and fetch the necessary data from the system. The I/O requests in the block traces were recorded using a 512-byte sector size, and the size of a cache block is 4KB, so a single request can span an arbitrary number of blocks. Thus, the replay application first converts the request into a list of all blocks it affects. For writes, since we are evaluating a write-through cache, we invalidate all cached blocks accessed by the write request until data are persisted to the backing store and then inserted into the tier-1 cache. If the write is not page-aligned, we also read the misaligned page from the cache or backing store into the tier-1 cache. The write is then asynchronously submitted to the backing store. For reads, the application checks whether all the pages required to satisfy the request are in the cache and fetches any missing ones from the backing store, using a single request if the pages are consecutive. If one or many blocks belonging to a block request require access to the backing store, requests to the backing store are made, and the request is added to the list of pending requests. A block request is considered complete when all pages related to the request are read from the cache and/or backing store, after which it is removed from the list of pending requests.

Backing Store Processor The backing store processor threads check for the completion of any asynchronous IO that was submitted to backing store. The backing store request contains metadata like the timestamp of submission and the ID of block request to which it belongs. The thread tracks the latency of the request to the backing store and updates the status of the block in the corresponding block request. If all the blocks related to the block request have been completed, the thread tracks the latency of the request and removes the request from the list of pending block requests.

4.2.3 Block Storage System

The block storage system consists of a cache and backing store.

Cache design We used CacheLib [21], an open-source caching engine, to implement the cache, which can be single-tier (memory-only) or two-tier (memory and SSD). The performance of a caching system depends on its configuration; in our work we focus on the following parameters: page size, replacement policy, cache admission policy, write policy, number of tiers, and tier sizes. We chose 4KB pages because that is the most common I/O size for modern storage devices and file systems [70, 111].

In all experiments, we used an LRU replacement policy, since it is widely used and has a convenient *inclusion property*: given the same input sequence and replacement policy, elements in a cache of size n would also be present in a cache of size $n + 1$. We used LRU in both cache tiers (however, see *Cache admission* policies below).

Tier sizes govern the sizes of tiers 1 and 2. Traditionally, both CPU and storage caches have been configured in a *pyramidal* arrangement where each successive (lower) tier is slower and larger than the one above [68]. Although pyramidal caches have convenient mathematical properties, we found that the conventional configuration is not always the best choice; for some workload and device combinations, it may be better to violate that arrangement and instead create a *non-pyramidal* hierarchy (see Section 4.3.5).

Cache admission policies control the actions after a miss or hit in each tier; they indirectly determine the overall latency for each tier in the multi-tier cache. Admission policies can be broadly classified as (i) *inclusive*, where any object admitted to tier 1 is also admitted to tier 2, and (ii) *exclusive*, where an object is admitted to tier 2 only when it is evicted from tier 1. In exclusive caches the object can be admitted to tier 1 either when there is a miss on both tiers, or when it is found in tier 2. In all our experiments, we focused on an exclusive admission policy to reduce the overhead of duplicated cache entries and to explore non-pyramidal configurations; an inclusive policy makes non-pyramidal caches impossible because by definition an inclusive second tier must be at least as large as the first. Thus, in our two-tier experiments every eviction from tier 1 is admitted to tier 2.

Because we used CacheLib to implement our caches, our admission and eviction policies are controlled by that package. CacheLib maintains a 4KB page size in tier 1, but tries to preserve SSD lifetime by using 16MB *regions* in tier 2. One tier-2 region is active at any given time and is used to collect 4KB pages as they are evicted from tier 1; those pages are written sequentially into the active region regardless of their placement in the address space, so that each 16MB region contains a mix of (relatively) recently referenced pages that may not share spatial locality. When the active region is full, it is moved to the head of the tier-2 LRU list, and the region at the end of the list is discarded and becomes the new active region.

A page hit in tier 2 causes the relevant page to be removed from its 16MB region and brought back into tier 1; this does not require any tier-2 writes because the metadata is kept in main memory. At the same time, that region is returned to the head of the tier-2 LRU list. This means that all of the other 4KB pages that were clustered with the newly promoted page will effectively be marked as having been referenced recently; this effect is reasonable because those other pages necessarily have similar last-reference times and thus have roughly similar temporal (though not spatial) locality. However, it does mean that the tier-2 cache only approximates true LRU behavior.

The *write policy* of a cache dictates how data is stored and moved between the various cache tiers and the backing store after a write. In our experiments, we used a *write-through* policy, which writes the data to tier 1 and asynchronously flushes them to the backing store (bypassing tier 2, in the multi-tier case); a request is marked complete when the asynchronous write has persisted. This ensures durability, which is critical in storage systems; in our case, it also maintains exclusivity. We use direct I/O to write to the backing store, thus bypassing the file system page cache.

Storage system We use Ubuntu 20.04 and Ext4 for all our experiments. The storage system uses `libaio` to submit asynchronous I/O requests directly to backing storage, avoiding the file system and OS page caches. Asynchronous requests do not block the program while the request is processed, allowing threads to process additional block requests while other I/Os are pending. We use `dd` to sequentially create a large file in the backing store to act as a disk; we submit I/Os directly to this file at the correct block offset. The size of the file might be insufficient for a workload where a large range of offsets are accessed; in that case we use the offset modulo the file size. However, as discussed in Section 4.3.2, this design limitation had a negligible effect on our analysis and conclusions.

4.3 Evaluation

We focus our evaluation on insights derived from trace replay and on the accuracy of our decision-tree model under varying constraints. We begin by presenting details of our test bed, workloads, and experiments. Then we describe the insights we derived from those experiments. To help system administrators maximize the benefits and avoid the pitfalls of MT caching, our investigation considers the following questions: **(1)** [Section 4.3.4] What is the overhead of adding a cache tier? What factors influence the overhead when a second-tier cache is added? **(2)** [Section 4.3.5] What combinations of tier devices and tier sizes improve throughput? Does the tier-2 cache have to be larger than the tier-1 cache to improve throughput compared to an ST cache? **(3)** [Section 4.3.6] How many of the evaluated MT caches were cost-effective, and for what relative cost of tier-1 and tier-2 caches? How does the relative cost of tier-1 and tier-2 caches affect the fraction of multi-tier caches we found to be cost effective? **(4)** [Section 4.3.7] How does an application parameter like queue size influence MT cache performance (measured by throughput)? Do insights regarding tier sizes and device combinations hold when the queue size is changed? **(5)** [Sec-

Name	CPU	Memory	T2 Device	Disk
c220g1	2 Intel E5-2630 v3 8-core CPUs	128GB ECC DDR4 1866MHz	Intel DC S3500 480 GB SSD	Seagate 10K RPM 1.2 TB HDDs
c220g5	2 Intel Xeon Silver 4114 10-core CPUs	192GB ECC DDR4 2666MHz	Intel DC S3500 480 GB SSD	Seagate 7.5K RPM 1 TB HDDs
r6525	2 32-core AMD 7543	256GB ECC DDR4 3200MHz	1.6TB NVMe SSD	Intel DC S3500 480 GB SSD

Table 4.1: CloudLab machines and specifications used for our experiments, from CloudLab's Web site.

tion 4.3.8] How does the request rate influence the throughput of an MT cache relative to an ST cache with the same tier-1 size? Do insights regarding tier sizes and device combinations hold when the request rate of a workload is changed?

Finally, in Section 4.3.9 we present the results of our evaluation of the decision-tree model and the criteria that the tree uses to predict, with high accuracy, whether or not to add a cache tier.

4.3.1 Physical Testbed Setup

We performed our evaluation on three different classes of bare metal servers in CloudLab; see Table 4.1 for details of their configurations. Machines c220g1 and c220g5 used the same SATA SSD tier-2 device, but differed in (i) the backing-storage device, Seagate 10K RPM HDD vs. a 7.5K RPM HDD, and (ii) the memory speed, 1866MHz vs. 2466MHz. The machine r6525 used a SATA SSD as its backing store, an NVMe SSD as tier-2 cache, and 3200MHz DDR4 memory. The SATA SSD used as a backing store by machine r6525 is identical to the SATA SSD used as a tier-2 cache by c220g1 and c220g5. The performance difference between the tier-2 cache device and the backing-store device is highest on machine c220g5, followed by machines c220g1 and r6525, respectively. We found that this difference has a significant impact on the performance of MT caches relative to ST caches.

4.3.2 Workloads

For our evaluation, we used the CloudPhysics traces as described in Section 2.3. Ideally, we would have liked to evaluate all 106 of those traces. However, replaying them all under various replay-rate and queue-depth settings for both single-tier

(ST) and multi-tier (MT) caches would have been prohibitively expensive (about 10× longer than the 7.3 compute years we have already consumed); more importantly, processing traces with similar features would not have necessarily revealed new insights. Therefore, we opted to select a subset of traces that were *representative* of the statistical distribution of the complete set.

To choose that subset, we used clustering to find similar groups of traces based on 184 predefined features summarized in Table 4.2. One machine-learning rule of thumb for handling the so-called "curse of dimensionality" is that, for each feature in a dataset, there should be at least 10 samples [10, 90]. However, we have 106 workloads and 184 features, so we used an autoencoder [52] to reduce the 184 features to 10 features. An auto-encoder is a deep-learning model that learns to recreate the data points from a fixed-size latent space. Formally, a latent space refers to an abstract multi-dimensional space containing features that we cannot interpret directly, but which encodes a meaningful internal representation of externally observed data points. Hence, learning a latent space is akin to dimensionality reduction and aims to capture the most important patterns required to learn and reconstruct the original data points. In short, the autoencoder maps the latent space in a continuous manifold (mathematical object that in each point appears to be flat [30]), ensuring that data points of the same cluster are mapped together. We used those 10 features to cluster the 106 traces using hierarchical clustering. The ideal number of clusters was identified using the elbow method [81], which selects the inflection point in the distortion curve and uses that as an indication that the underlying model fits the data points. Distortion measures the quality of the clustering process, low distortion means that the data points tend to be placed near the centre of the cluster. A distortion curve represents the distortion values per number of clusters considered. After applying the elbow methods, we got 14 clusters. For each cluster, we selected the trace closest to the cluster's center to represent that cluster.

Feature	**Description**
Request Count	Read/write block request count
Total I/O Requested	Total read/write I/O in bytes requested
I/O Range	Difference between the maximum and minimum offset accessed
Sequential Count	Number of sequential block requests
Total I/O Misalignment	Total bytes that are not page aligned
Working Set Size	Read/write working set size
Block Request Size	Rd/Wr block request sizes (percentiles)
Jump Distance	Percentiles of difference in end offset and start offset of consecutive requests
Scan Length	Percentiles of length of consecutive writes or cold read miss
Inter-arrival Time	Percentiles of inter-arrival time between block requests
Page popularity	Read/write page popularity (percentiles)

Table 4.2: Workload features used to group workloads.

Using this procedure, we selected the following 14 traces from the CloudPhysics set: w11, w14, w18, w20, w32, w36, w46, w47, w53, w54, w68, w81, w82, and w98. Table 4.2 displays a few characteristics relevant to this work for an illustrative subset of these traces. In the end we had to drop workload w54 from our evaluation because it had block requests to offsets much larger than 1TB, which were out of range for the HDDs available in the CloudLab machines we were able to use. Therefore, our final list includes 13 representative traces.

To handle out-of-range block offsets, we had three choices: we could wrap the offsets using modulo arithmetic, ignore the requests with the large offsets, or eliminate the offending traces entirely. We chose to eliminate trace w54, because it had offsets of up to 36TB and more than 98% of them were greater than 1TB. However, we kept trace w53 but wrapped its offsets for all experiments involving this workload; it was the only other trace with offsets over 1TB: it had offsets up to 1.6TB but only 8% of them exceeded 1TB. We do not feel that using this wrapping policy for a single trace substantially affects our analysis or conclusions.

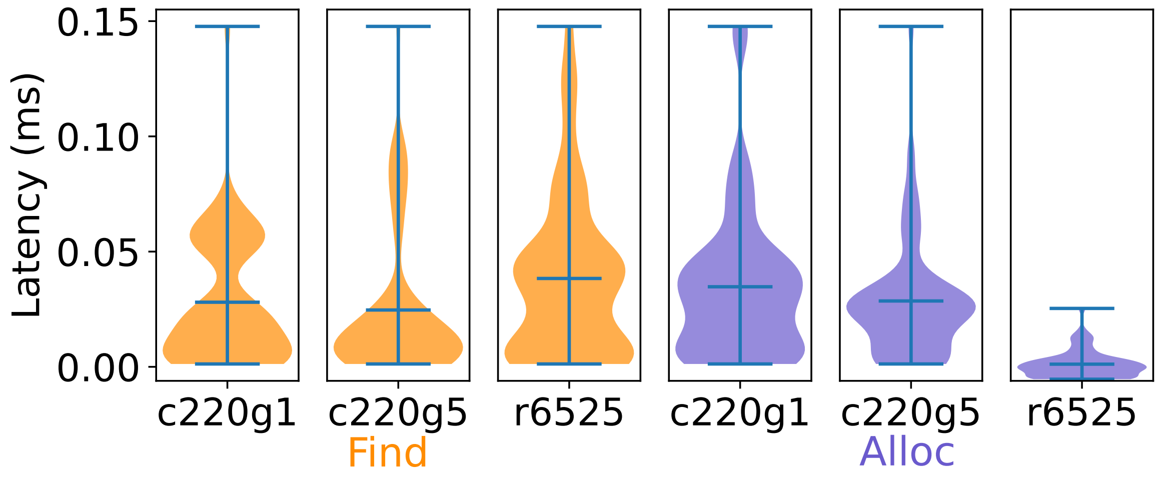


Figure 4.2: The increase in FIND (left) and ALLOC (right) latency when adding a tier-2 cache, for all machine classes.

4.3.3 Experimental Methodology

We run 18,594 experiments on three different classes of machines, using the 13 representative workloads, and various cache configurations, totaling 7.3 years of compute time. We ran three iterations of each experiment, resulting in 6,198 experimental sets. We use the mean of each metric in our evaluations. The results were fairly stable; the mean percentage difference in throughput in the iterations was merely 0.1%.

4.3.4 Multi-Tier Overhead

In our experiments, we tracked the latency of each FIND and ALLOC CacheLib API call and computed the latency percentiles. ALLOC allocates space in the cache when a new item has to be inserted; FIND checks whether an item is in the cache and returns it if found. The latency of FIND and ALLOC inherently increases when we add a second cache tier: FIND because both tiers must be searched, and ALLOC because data evicted from tier 1 must be moved to tier 2 rather than simply being discarded. Figure 4.2 shows the increase in latency that we observed when we added a tier-2 cache, in 434 experiments with identical parameters across three machine classes listed in Table 4.1. The first three plots show the distribution of increases in FIND latency when a tier-2 cache was added, and the last three show the distribution of increases in ALLOC latency. The width of the violin plot at each Y coordinate represents the number of data points at that value. We can see that the difference in FIND latency across machine classes was smaller than that for ALLOC latency, and that the latency of ALLOC is more sensitive to the machine class. Machine r6525, which has the fastest DDR4 memory, had much smaller ALLOC overhead because the internal data structures maintained in DRAM can be updated much more quickly. Machine c220g1 had higher FIND and ALLOC overheads because it has the slowest DDR4 memory of the three. This shows that a machine with faster memory will have lower overhead when a tier-2 cache is added. However, machine r6525 had the worst FIND latency because it processes block requests at a much higher rate than machines c220g1 and c220g5, since machine r6525 has the fastest memory (3200MHz) and backing store (SATA SSD). A workload that puts pressure on the application queue, which leads to high queuing time when it is replayed on machines c220g1 and c220g5, might not stress machine r6525 as much because cache hits and misses are processed much faster on that platform. Processing requests faster leads to serving more FIND and ALLOC requests in a shorter time, which increased the mean latency of FIND on machine r6525 even though it has the fastest memory. However, we can see that the low overhead of ALLOC on machine r6525 made up for the high overhead in FIND, and overall, machine r6525 has the lowest overhead compared to c220g1 and c220g5.

For a tier-2 cache to improve performance, the overhead introduced by adding it must be offset by the latency gain from tier-2 cache hits. To highlight that fact, Figure 4.3 shows the latency differences between the tier-2 cache and the backing storage, for both reads and writes. The largest read-latency difference appeared in c220g5 because it had the slowest backing store (a 7.5K RPM HDD). The larger latency gain from tier-2 read hits and the lower overhead from adding a tier-2 cache made c220g5 a better fit for improving performance by adding a tier-2 cache than either machine c220g1 or r6525. Machine r6525 uses a SATA SSD as its backing store

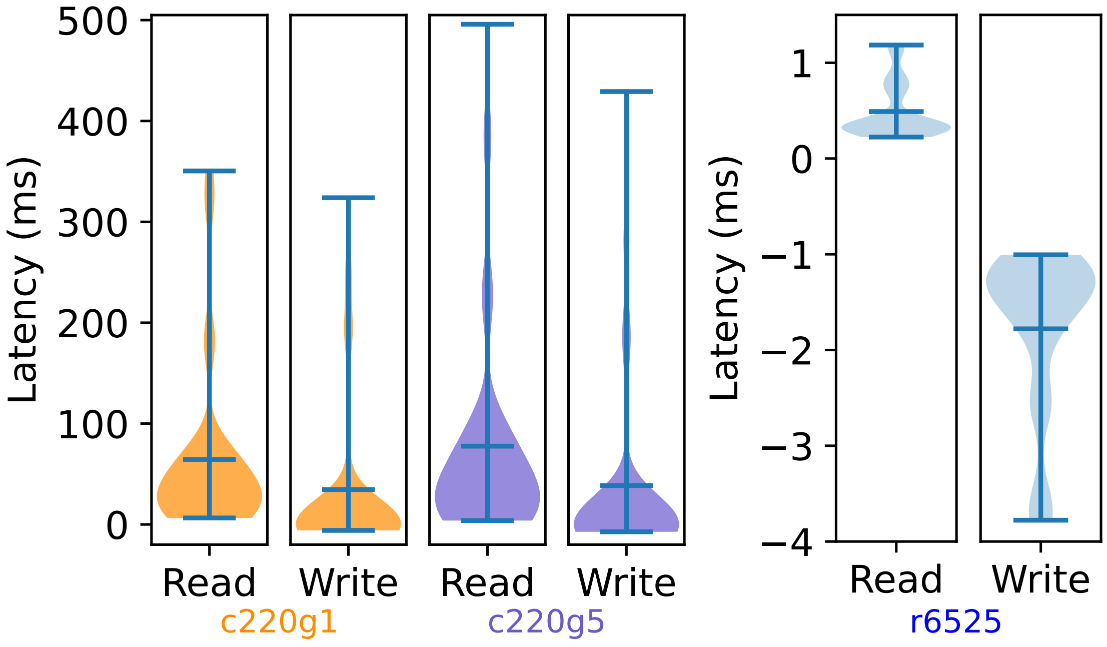


Figure 4.3: The differences in read and write latencies of tier-2 caches and backing store across three machine classes, c220g1 (left), c220g5 (middle), and r6525 (right).

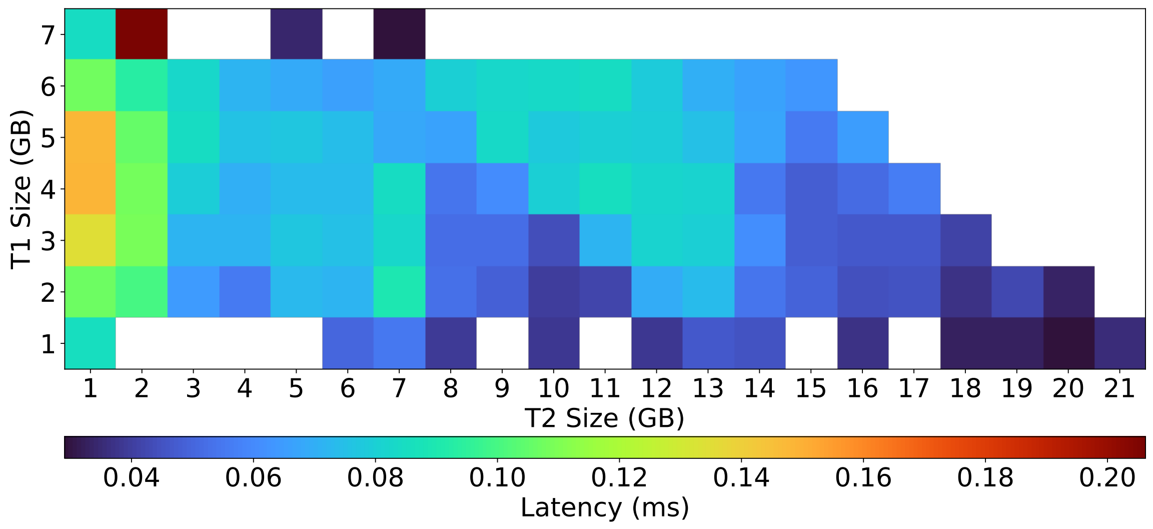


Figure 4.4: Heatmap showing the means of average latency values of experiments with different tier sizes.

and an NVMe SSD for the tier-2 cache, so its tier-2 hits result in the smallest gain in read latency, making it poorly suited to multi-tier caching. That conclusion is further reinforced by the fact that the write performance of the NVMe SSD suffers from high load when it is used as a tier-2 cache, to the point where it performs *worse* than the SATA SSD backing store as shown in Figure 4.3.

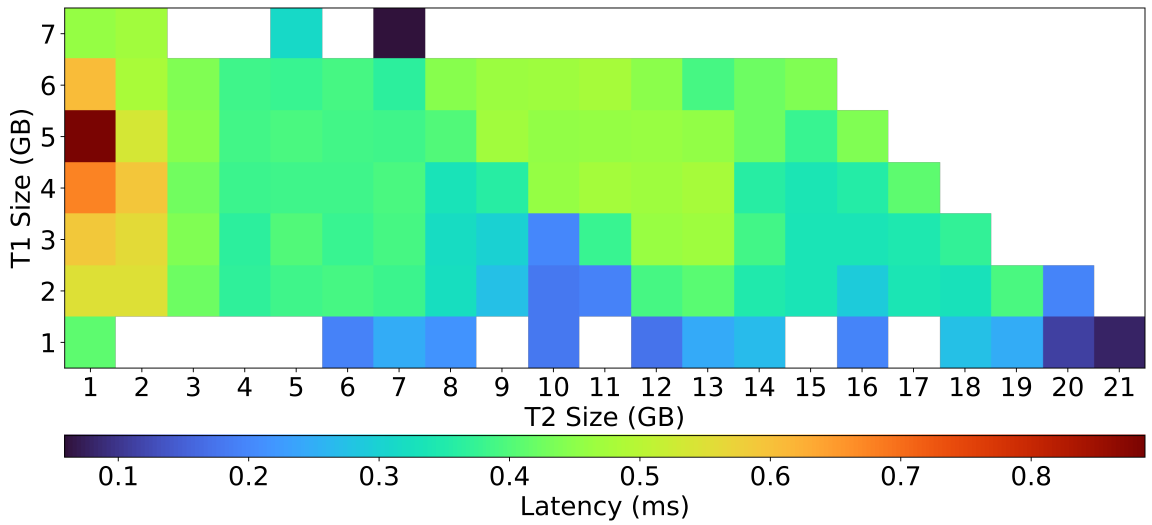
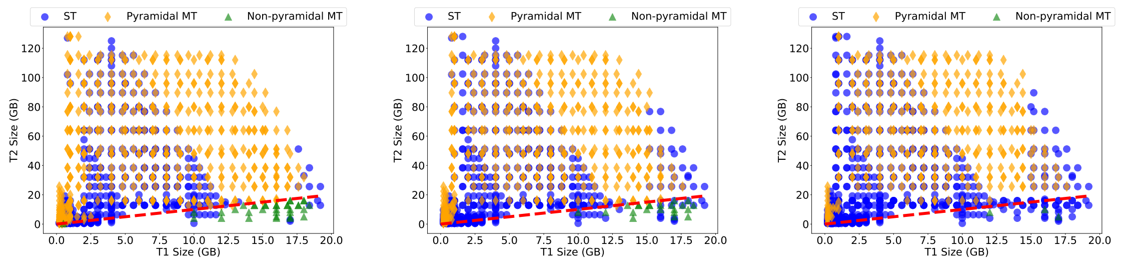


Figure 4.5: Heatmap showing means of p99 latency values of experiments with different tier sizes.

Figures 4.4 and 4.5 show, respectively, heatmaps of the means of (i) mean and (ii) the p99 latency of the sum of FIND and ALLOC calls across different tier sizes. For example, in Figure 4.4 we can see that small tier-2 sizes such as 1GB and 2GB see the largest sum of FIND and ALLOC latency (dark box = high latency), while large tier-2 sizes see comparatively lower latency (light box = low latency). Unsurprisingly, we can see that when both cache sizes are small, latencies tended to be high—especially the p99 values. Overall, we found that specific device parameters (for memory, tier 1, tier 2, and backing store) and tier sizes all influenced the overhead and hence any performance gain when a tier-2 cache was added. We can also see that a fast tier-1 cache and a slow backing store were both favorable for MT caching; this is because a fast tier-1 cache ensures that the additional internal metadata structures needed to support an added tier-2 cache can be accessed more quickly, and a slow backing store implies that there will be a large latency benefit from tier-2 cache hits. In summary, the larger the performance gap between the tier-1 cache and the backing store, the likelier it was that adding a tier-2 cache would improve performance.



(a) Improvement $\geq 10\%$ (b) Improvement $\geq 20\%$ (c) Improvement $\geq 30\%$

Figure 4.6: The best cache types for different tier sizes and minimum desired performance improvements (10%, 20%, and 30%). ST means that not having a second tier at all was best. When a second-tier cache improved performance, we compared the sizes of the tiers and classified them as pyramidal if the size of tier 2 was larger than that of tier 1, and as non-pyramidal otherwise.

4.3.5 Pyramidal Multi-Tier Caches

As shown in Figure 1.1, a *pyramidal* cache follows the convention that each successive tier is larger and slower than the preceding one, while a *non-pyramidal* cache does not strictly follow that arrangement. In this work, we examined non-pyramidal configurations in which lower tiers could be smaller than higher ones; we did not investigate violations of the speed hierarchy.

We evaluated a total of 6,198 caches, of which 1191 were ST caches, 3946 were pyramidal MT caches, and 1061 were non-pyramidal MT caches; we evaluated them across 3 machine classes, using a diverse set of tier sizes and application parameters. We evaluated 1,859 caches on machine c220g1, 564 on c220g5, and 3,775 caches on r6525. The difference in the number of evaluated points between machine classes is largely based on the number of machines of each machine class type available in CloudLab, its availability over time, and the speed at which the machine can replay the workload. The figures referred to in this section use data from machine c220g1 because it completed the most experimental points; c220g5 has fewer points because of the smaller number of evaluations conducted. Machine r6525 had only a few instances where MT caches improved performance even though the evaluation set was large.

The splits of percentages in MT-vs.-ST comparisons that favored ST, pyramidal

MT, and non-pyramidal MT for the three machine classes are shown below:

Machine:	**c220g1**	**c220g5**	**r6525**
# Comparisons	1,351	402	3,017
Favored ST	38%	45%	97%
Favored Pyramidal MT	53%	31%	2%
Favored Non-Pyramidal MT	9%	24%	1%

The number of comparisons is lower than the number of caches evaluated for each machine because some of the caches evaluated were ST caches, we needed to compare to MT caches. We see that favorable non-pyramidal configurations exist even for machine r6525, which generally did not favor adding a tier-2 cache. The mean increases in throughput when pyramidal and non-pyramidal MT caches improved performance over an ST cache were 81% and 90%, respectively, for machine c220g1, 68% and 176% for machine c220g5, and 33% and 23% for machine r6525.

Thus far, we considered the smallest increase in throughput as an improvement. However, real world users would consider workload variation and cache-tuning costs, thus preferring a minimum performance-improvement threshold. Figure 4.6 shows, for machine c220g1, whether a multi-tier cache with given tier-1 and tier-2 sizes improved performance by more than a given percentage threshold compared to its single-tier counterpart. We see that the number of multi-tier configurations shrank as the performance threshold increased. Interestingly, the tier sizes change from performing better in MT configurations to performing better with just a single tier as we go from left to right in Figure 4.6; these changes appear at both low and high tier-1 sizes. Configurations in the region with tier-1 sizes of 9–15GB and tier-2 sizes greater than 20GB stayed pyramidal MT even at high performance thresholds in Figure 4.6. This aligns with our observation in Section 4.3.4 that MT caches with small tier-1 sizes have higher overhead and are less likely to outperform MT caches with medium and large tier-1 sizes. However, MT caches with large tier-1 sizes absorb most of the cache hits in tier 1, limiting the number of tier-2 cache hits and thus reducing the performance improvement available from adding a tier-2 cache; thus, multi-tier caches

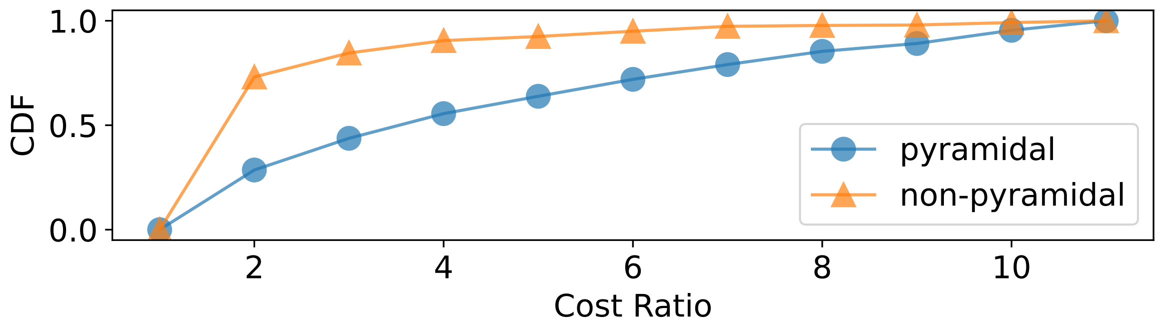


Figure 4.7: The CDF of cost ratios for machine c220g1. A low ratio means that the MT cache was more cost-effective than the ST cache it was compared to. The CDF shows the percentage of pyramidal and non-pyramidal MT caches found to be more cost-effective than ST ones, for varying relative tier-1 and tier-2 costs. For example, around 70% and 90% of pyramidal and non-pyramidal configurations, respectively, were more cost-effective when the tier-1 cache cost 6× more than the tier-2 cache.

with large tier-1 sizes become less preferred as the required improvement increases.

4.3.6 Lowering Costs

Even if adding a second-tier cache improves performance, that does not mean that it also reduces costs. For example, it could be cheaper to increase the size of the single-tier cache than to add a second tier. For example, consider an MT cache, M, with a 5GB first tier and a 20GB second tier, that outperforms a 5GB ST cache, $S1$. Now we compare M with a 10GB ST cache $S2$. If M also outperforms $S2$, it means that it may be cheaper to add a 20GB tier-2 cache than to increase the first tier from 5GB to 10GB. Thus, if the cost of 20GB of tier 2 equals the cost of adding 5 GB to tier 1, then it is better to purchase the second tier than to expand the first; we thus call the ratio of the two sizes the *cost ratio*, which in this example would be 4. As long as the tier-1 cache costs 4× the tier-2 cache (or more), adding the tier-2 cache and using configuration M would be cheaper than increasing the tier-1 size and using configuration $S2$.

We considered scenarios where a storage operator has to choose between increasing the size of the tier-1 cache or adding a tier-2 cache. To evaluate such scenarios, we compared each MT cache that outperformed its ST counterpart to an ST cache with a larger tier-1 size. If the MT cache outperforms an ST cache with a larger tier-1 size, then adding a tier-2 cache is cost-effective for some values of the ratio of cost between tier-1 and tier-2 devices. We collected the cost ratios for all such comparisons and computed a CDF; Figure 4.7 shows that CDF for both pyramidal and non-pyramidal multi-tier caches on machine c220g1. Machines c220g1 and c220g5 had similar CDFs when we compared experiments using identical parameters (queue size, tier-1 size, tier-2 size, and replay rate) but we report results only for machine c220g1 because we have twice as many data points for it than we have for c220g5. We excluded machine r6525 because in most cases it did not offer improved MT performance compared to single-tier caches. The current market cost of the tier-2 cache device used in c220g1 and c220g5 is approximately 6× cheaper than RAM [14, 11, 12]. We see that for c220g1, non-pyramidal configurations are more likely to be cost-efficient; we observed similar results for c220g5. Assuming that the cost ratio is indeed 6×, around 30% of pyramidal MT caches had cost ratios of 6 or above, thus at current prices, around 30% of the pyramidal MT caches would be less costly than ST versions. Similarly, we found that around 10% of non-pyramidal MT caches would be cheaper at current market rates.

Finally, we address cost reduction with respect to tier sizes, as shown in Figure 4.8. The lower the MT cache's cost ratio, the more cost-effective it is. Thus, we see that a small tier-1 cache backed up by a large tier-2 cache (*i.e.*, pyramidal configurations) was the most cost-efficient choice for MT caches. Using a large tier-1 cache with a fairly small tier-2 size performed poorly and was not cost effective. While non-pyramidal configurations could also perform well, as shown in Figure 4.7, Figure 4.8 shows that they were not as cost effective as pyramidal MT caches. And although small tier-2 sizes could improve performance, they were not cost-effective. Instead of adding a small tier-2 cache, we found that it was better to stick with an ST cache and slightly increase its tier-1 size. We found that either a pyramidal or a non-pyramidal

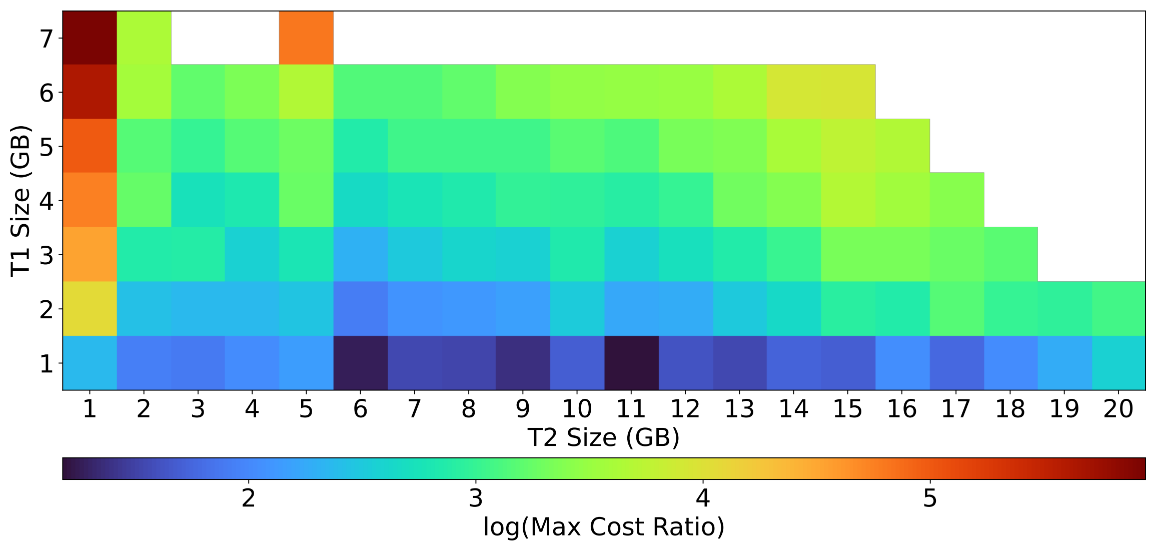


Figure 4.8: Heatmap of mean cost ratio for multi-tier caches with different tier sizes that improve performance when compared to a single-tier cache with the same tier-1 size.

cache could be the least expensive option; but we also found more instances where pyramidal caches were less costly, especially those with a small first tier and a large second tier.

4.3.7 Queue Depth

The queue depth, which is the maximum number of pending block requests that can exist in the application at any given time, impacts the load on the system while replaying the trace. If the application queue is full, the application pauses further block requests until queue space is available. The queue depth thus represents the maximum pressure placed on the block-storage subsystem when replaying a workload.

The ideal queue depth is determined by the characteristics of the machine and the workload. We changed the queue depth from 128 to 256 for machines c220g1 and c220g5 and from 64 to 128, 256 to 512, and 512 to 1024 for machine r6525. We prioritized evaluating larger queue depths for machine r6525 because it uses faster devices than c220g1 and c220g5 for both tier 2 and the backing store (Table 4.1) and hence could handle more pressure. Our goal was to validate that the insights in Sections 4.3.5 were not limited to the particular application parameters that we used, but rather were applicable across a diverse set of parameters. We also wanted to evaluate the queue depth's influence on the performance of MT caches and its relative performance to ST caches.

We evaluated 45 MT caches for machine c220g1, of which 11 under-performed ST; among the 34 that performed better, 11 were non-pyramidal and 23 were pyramidal. We used both read-heavy workloads (w82 and w77) and write-heavy ones (w47 and w97). When the queue depth was increased from 128 to 256, 6 MT caches that improved performance over an ST cache with queue depth of 128 no longer did so at a queue depth of 256. This shows that queue depth can impact whether or not a tier-2 cache improved performance.

MT caches running read-heavy workloads saw a greater increase in throughput: 3.3% compared to 0.3% for write-heavy workloads. The relative throughput of MT and ST caches also saw a larger improvement of 2.8% for read-heavy workloads, compared to 0.13% for write-heavy workloads. This was because increasing the queue depth for write-heavy workloads increased the pressure on the backing store and the tier-2 cache, since more write requests caused more tier-1 evictions and write requests to the backing store. Conversely, increasing the queue depth of read-heavy workloads also increased pressure on the tier-2 cache and backing store, but the pressure on backing store was minimized by read hits from tier-2 cache.

We observed similar behavior with machine c220g5 when we changed the queue depth from 128 to 256. We evaluated 21 MT caches, of which 9 did not improve performance over an ST cache at queue depth 128; the 12 that did were evenly split between pyramidal and non-pyramidal. When the queue depth was increased from 128 to 256, the throughput of write-heavy workload decreased by 0.8% but it increased by 15.2% for read-heavy workloads, and the relative throughput of MT and ST caches decreased by 0.6% for write-heavy workloads but increased by 2.23% for read-heavy

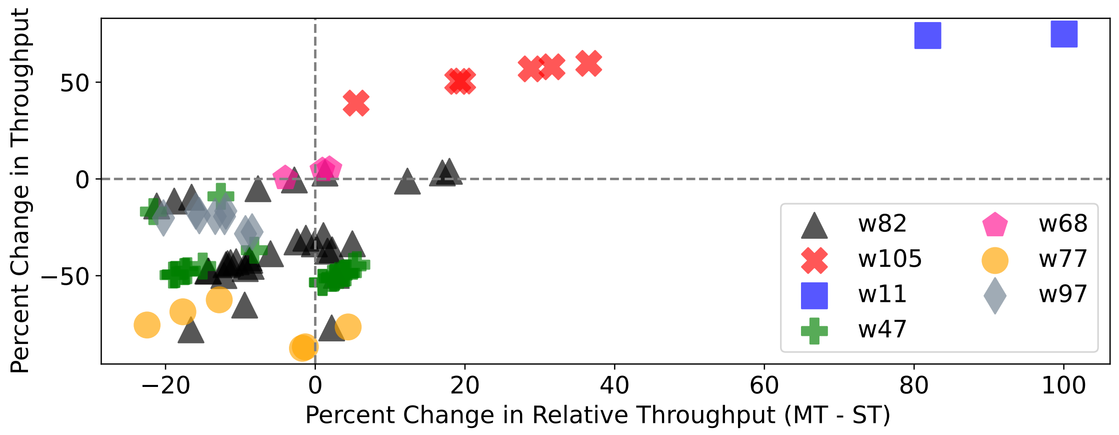


Figure 4.9: The influence of changes in the replay rate on the throughput of MT caches and their throughput relative to ST caches. MT caches with $x > 0$ are those whose throughput improved when the replay rate was increased from 1× to 100×. MT caches with $y > 0$ saw a throughput increase relative to an ST cache with the same tier-1 size.

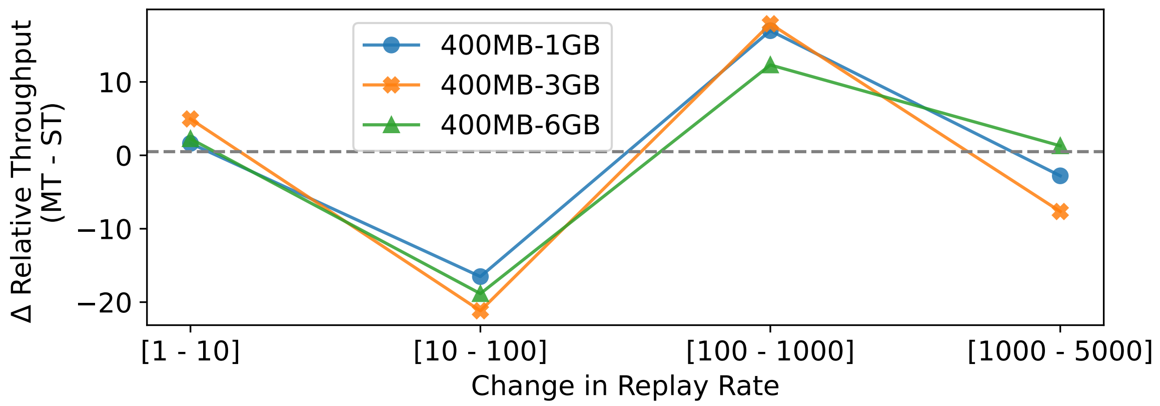


Figure 4.10: The change in relative throughput of 3 MT caches replaying workload w82 in machine c220g1 when the replay rate was increased. For a given workload and tier-1 size, increase in replay rate favors MT caches up to a point and then starts to deteriorate.

workloads. For machine r6525, however, a change in queue depth did not have an effect on whether an MT cache improved performance over ST, nor did it cause a significant change in throughput.

4.3.8 Replay Rate

We tuned the replay rate to evaluate performance under different request rates while preserving the spatial properties of a workload. This allowed us to do a “what-

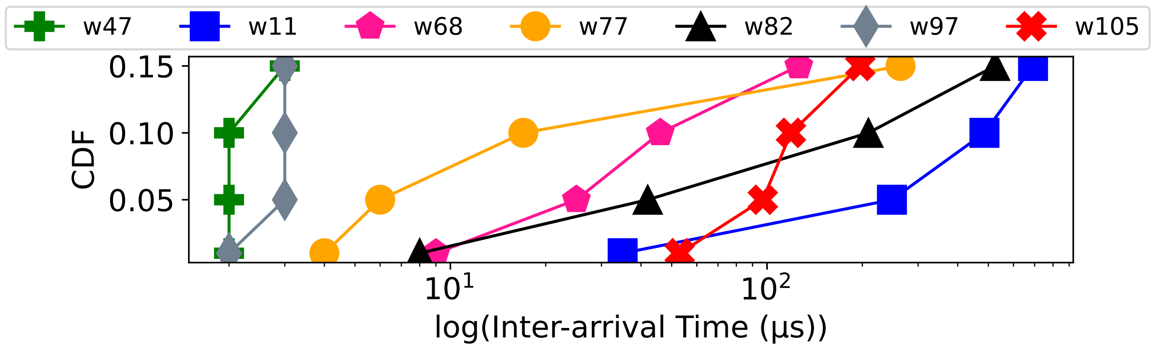


Figure 4.11: Lower percentiles (1, 5, 10, 15) of inter-arrival time of block requests for each workload. Lower inter-arrival values impact MT cache performance by increasing the stress on the storage system.

if" analysis of the effect of request rate and increased stress on MT cache performance. We used replay rates ranging from 1× to 100×. Although we evaluated intermediate values, for brevity we report only on the two extremes.

We replayed 68 MT caches on machine c220g1, 17 of which under-performed ST; among the 51 that performed better, 37 were pyramidal and 14 were non-pyramidal at a 1× replay rate. We used five read-heavy workloads (w82, w105, w11, w77, w68) and two write-heavy workloads (w97, w47) to illustrate these findings. The workloads also varied in their inter-arrival times between block requests (see Figure 4.11). Workloads w11 and w105 stand out as having larger inter-arrival time values at lower percentiles values of 1% and 5%. This means that there are fewer request-rate spikes and the pressure is lower on the system.

We found that non-pyramidal MT caches could improve their performance compared to ST caches at both 1× and 100× replay rates. Compared to their respective ST caches, 10 MT caches switched from under-performing to improving performance, and 10 MT caches switched from improving over ST caches to under-performing. This shows that the request rate is important in determining whether or not to add a tier-2 cache—that decision changed in a total of 20 out of 76 (26%) MT caches evaluated. All the MT caches evaluated on machine r6525 under-performed ST caches; that did not change when the replay rate was increased to 100×.

Next, we evaluated whether increasing the replay rate improved the throughput of MT caches and their relative throughput compared to ST caches. Figure 4.9 shows the change in throughput of MT caches on the Y axis and, on the X axis, how the difference in throughput between MT and ST caches changed when the replay rate was changed from 1× to 100×. The upper-right quadrant, for example, represents the MT caches where both throughput and relative throughput improved when the replay rate was increased. The lower-right quadrant represents those caches where the throughput decreased but the relative throughput rose. Overall, we see that different workloads reacted differently to increases in replay rate. However, workloads w11 and w105 stood out: even their smallest inter-arrival times were much larger than those in other workloads as shown in Figure 4.11. Consequently, they showed the largest improvements in throughput and relative throughput difference. The two MT caches evaluated for w11 already outperformed an ST cache by 70% and 90%; when the replay rate was increased to 100×, those MT caches outperformed the ST versions by 152% and 190%, respectively. However, the 4 MT caches for which we replayed workload w105 had 18.5% lower throughput than ST caches at a 1× replay rate, but at 100× they had 5% *higher* throughput than ST.

These experiments showed that increasing the replay rate favored MT caches over ST caches if the inter-arrival time of the workload was large enough; but if the workload's inter-arrival time is already small, then a further increase in replay rate favored ST caches. However, it does *not* mean that if the replay rate continues to increase, the relative performance difference between ST and MT caches continues to improve. Figure 4.10 shows that relative throughput between MT and ST caches increased when the replay rate increased from 100× to 1,000× but reduced when replay rate was further increased to 5,000× for machine c220g1. We find that MT caches do not favor low request rate, as the backing store is underutilized; the gain from tier-2 read hit is minimal. As the request rate increases, the backing store gets

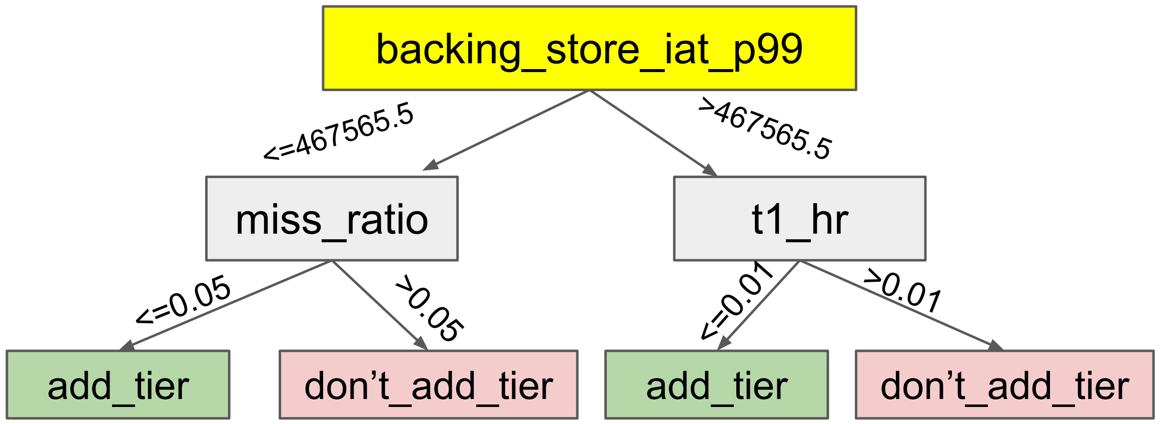


Figure 4.12: Example decision tree to predict whether or not to add a tier-2 cache, with accuracy of 90%, in machine c220g5.

busier and the performance gain per tier-2 cache hit increases. However, if the request rate continues to grow, it now starts to *hurt* the performance of the tier-2 cache, and the performance gain per tier-2 cache hit reduces. Thus, low request rates do not favor MT caching, nor do very high request rates. There is a sweet spot where the backing store is stressed enough, while the tier-2 cache is not, similar to that seen in Figure 4.10. This sweet spot can differ according to the devices used in the machine.

4.3.9 Decision Trees

We evaluate the machine learning model for two types of machine: c220g1 and c220g5. Our goal is to achieve high accuracy with decision trees, to reveal the important features that influence whether adding a tier-2 cache improves performance. We evaluate decision trees under different minimum performance improvement thresholds to show that they work for different requirements. A change in this threshold alters the binary values that represent whether or not to add a cache tier. For instance, an MT cache might yield a performance improvement of 2%; when we increase the improvement threshold from 0% to 3%, the binary target variable for this MT cache will change from 1 (add a tier) to 0 (do not add a tier).

Model Performance On average across performance thresholds of 0%, 3%, and 5%, we achieved 88% accuracy, 87% weighted f1 score, 88% weighted precision, 88% weighted recall for machine c220g1; and 90% accuracy, 88% weighted f1 score, 89% weighted precision and 87% weighted recall for c220g5. We describe one such decision tree in Figure 4.12 for the improvement threshold of 0%. It uses the 99% percentile of IAT of requests to backing store as the first splitting feature. If the value of inter-arrival time to backing store is lower than 467,565.5μs then it decided to add a cache tier if the miss ratio is very low (0.005). So, the decision tree implies not to use a tier-2 cache if the request rate of workload is high and there are a lot of misses. If the IAT is higher than 467,565.5μs which implies a low request rate, then we add a cache only if tier-1 hit rate is low. This implies that workloads with low request rate require larger tier-2 hit rate to gain performance and make up for the overhead of adding a tier-2 cache.

We generated many decision trees that use different set of features that gain high accuracy; however, all decision trees consistently used both hit rate and some temporal feature—validating our findings of these two features' importance.

4.4 Discussion

Replacement Policy In this work, we focus on the LRU replacement policy. Although LRU is one of the most widely used replacement policies, other replacement policies along with variants of LRU provide different benefits. Replacement policies cannot be simply discarded on the basis of hit rate analysis, as some replacement policies have lower overhead, which could outperform replacement policies with higher hit rate and overhead [105, 110]. Alternative replacement policies need to be evaluated separately for tier-1 and tier-2 cache, as the former is designed to work with DRAM whereas the latter is designed around flash devices.

Tier-2 Optimization The tier-2 cache is designed and implemented to support flash devices. It contains numerous configuration parameters that are set to reduce write amplification. Our current evaluation also admits all tier-1 evictions to tier-2. There may be scenarios in which selectively admitting items to the tier-2 cache could reduce pressure on the tier-2 device, improving performance even if it causes a slight dent in the hit rate. The potential for MT caching is higher than what we have even seen if we start tuning the tier-2 cache parameters. The optimal tier-2 parameters will depend on the type of device and the workload. Some example parameters of tier-2 cache parameters are as follows:

Admission rate Tier-2 cache can be configured to randomly admit a limited percentage of tier-1 cache evictions. Depending on the potential tier-2 cache hits and the rate of tier-1 evictions, tuning the admission rate of the tier-2 cache can help improve overall performance.

Number of clean regions To avoid frequent small writes to flash that degrades the lifetime of the device. Cachelib batches blocks that are evicted from tier-1 cache and arranges them into regions sized 16MB by default before flushing them to the tier-2 flash cache. It maintains a number of clean regions in memory to admit tier-1 evictions before flushing them to tier-2 cache. If the rate of tier-1 evictions is high, it could benefit to have more clean regions available to absorb tier-1 evictions while regions are being flushed to the tier-2 flash device. The size of the region is also another parameter that can be adjusted. A larger region size would reduce the frequency of writes, but each flush would take longer to finish.

4.5 Related Work

Our work builds upon prior research in several different areas, including hierarchical storage systems, MRC construction, workload characterization, and machine-learning techniques.

Hierarchical storage systems Research on hierarchical storage systems with multiple caching levels or tiers has primarily focused on solutions tailored to specific domains, such as virtual machines [107, 106, 62, 75], cloud storage [24], and file systems optimized for multiple tiers [46]. Recent work on *non-hierarchical* caching [101] has also found that with some modern storage devices, it is possible to improve aggregate performance by shifting excess load from a performance device (*e.g.*, an Optane SSD) to a capacity device (*e.g.*, a flash-based SSD). In contrast, we analyze a broad range of cache properties and sizes, including non-pyramidal configurations, that are generally applicable to many environments.

Miss-ratio curves Reuse-distance analysis is a powerful technique for characterizing the temporal locality of workloads, often summarized using miss ratio curves (MRCs) [60]. Modern spatial sampling techniques, including SHARDS [96], miniature simulation [97], and AET [40] have dramatically reduced the time and space complexity of MRC construction, enabling practical applications in online systems. Zhang *et al.* [109] considered two-tier hierarchies with asymmetric read/write performance and extended sampling results for MRCs to accelerate simulations.

To model multiple cache tiers, eMRC [57] extended single-tier MRCs to multi-dimensional miss-ratio surfaces. However, the eMRC algorithm requires convexity, which limits its applicability to modeling multi-tier cache systems that employ cliff removal, which is not performed by any production systems. To model multi-tier throughput, eMRC also assumed that performance improves linearly with hit ratio. In contrast, we demonstrate that real-world performance is significantly more complex, depending on factors such as request rate and device-specific characteristics that are not reflected in hit ratios.

Workload characterization Understanding the characteristics of workloads is critical for evaluating, designing, or optimizing any storage solution. Researchers often analyze individual traces in an attempt to extract some distinguishing set of features (*e.g.*, the read-to-write ratio) [51, 59, 80, 17, 102, 25, 94]. Machine-learning techniques are popular for identifying new features or evaluating the importance of a pre-determined feature set [73, 72, 65, 83, 87, 59, 17]. These features can then be used for workload classification [31, 73, 65, 83, 80, 98, 17] and detecting workload phases or access patterns [72, 83, 69, 16, 73]. Li *et al.* developed Metis, a framework that uses customized Bayesian optimization to auto-tune cloud configurations and reduce tail latencies [51]. Salkhordeh *et al.* used online workload characterization to classify a workload as one of five classes, and then determined the optimal cache configuration based on that class [80].

Machine Learning in Systems Various statistical and ML algorithms showed that they can make beneficial tuning decisions [8, 7, 9]. Several applications to caching, including LRB [85], Parrot [53], and Hawkeye [44], have reduced the gap with Belady's optimal MIN algorithm [18]. Although MIN relies on knowledge of future references and is therefore unrealizable, these approaches train predictors to approximate it using techniques such as gradient boosting machines [34] and imitation learning [78]. However, as we have demonstrated, hit rates alone are not a reliable metric for multi-tiered caches, since device characteristics impact the overall request latency [36, 50].

Other research has developed new ML-based cache-replacement policies. Cacheus [77, 91] switches dynamically between various "expert" replacement policies using reinforcement learning. GL-Cache [104] demonstrated that fine-grained learning approaches often incur significant overhead that can result in lower overall throughput, and developed an algorithm that operates on larger groups of references to amortize these costs.

4.6 Conclusion

Multi-tier caching provides opportunities to optimize the cost and performance of some storage systems, but designing them is a complicated problem with a vast configuration space. We developed a framework that replays block traces and collects many performance metrics for both single- and multi-tier caches. We used our framework to analyze over 100 production traces, and conducted thousands of experiments on a representative subset of 13 traces, consuming 7.3 compute years. We trained a decision-tree model to determine whether adding a tier-2 cache to a single-tier system can improve performance, and achieved 84% accuracy for machine c220g1 and 90% for c220g5, showing that these tools can be effective for this predictive task.

Our experiments provided valuable insights. Surprisingly, non-pyramidal two-tier cache configurations can be more cost- and performance-efficient than single-tier caches. We discovered that multi-tier caching is typically useful for moderately-sized caches: adding a tier is often not beneficial when tier 1 is large, as the tier-2 cache is unable to provide enough hits to offset its additional overhead. Conversely, small cache sizes in both tiers introduce similar overhead but little performance gain. In addition, the temporal properties of a workload, such as the request rate, can significantly affect performance in ST and MT caches, and the choice between whether an ST or MT cache would be be preferable.

Finally, we found that it was not sufficient to characterize multi-tier cache performance using simple approaches such as miss ratios. In real-world experiments, complexities such as the non-sequential behavior of device queues, flash translation layers, and HDD performance variations led to a disconnect between those metrics and the observed behavior of a multi-tier cache.

Chapter 5 BlkSample: Sampling for Block Storage Traces

In this chapter, we present the design and implementation of BlkSample, a framework for sampling block requests that create sample traces that are representative of the full workload. Tracing is used by system administrators to tune systems, identify performance bottlenecks, and analyze workloads. Although tracing is very helpful, the overhead, both in time and space, can be substantial enough for system administrators to avoid it. Modern block storage workloads are intensive with a high request rate. Tracing such workload, even for a short period of time, can generate a block trace which consumes a large space and is computationally expensive to analyze.

Sampling is a useful technique to alleviate the overhead of tracing. If we sample block requests received by a block storage system instead of tracing every block request, we reduce the size of the resulting trace, which reduces the storage and computational requirements of the analysis. An example where sampling has been used successfully is in the generation of Miss Ratio Curves (MRC) [96]. Accurate MRCs were generated using sampling rates as low as 1%. Although the miss ratio has been used as a proxy for performance, we showed in Chapter 4 that it can be misleading when used in isolation. Block workloads with identical MRCs can have substantially different performance if they differ drastically in other features. Features such as the mean read/write request size, the mean read/write IAT, the mean read/write misalignment, and the write ratio are integral along with the MRC of the workload. BlkSample is a framework to create sample block traces from a stream of block requests such that the resulting samples are representative of a diverse set of features, not just MRC. The small size of the sample coupled with the low error in samples generated by BlkSample reduces the computation and storage overhead of block trace analysis.

We evaluated BlkSample with a diverse set of workloads. Compared to random spatial sampling [96], BlkSample generates accurate block trace samples that also show better replay performance. BlkSample samples regions instead of individual blocks and processes sample traces using workload features collected during sampling.

The following are the main contributions of this chapter.

1. We show that sampling with larger granularity leads to a significant improvement in sample quality compared to random spatial sampling.

2. We introduce a post-processing algorithm that further reduces the sampling error by using features generated from eight counters updated during sampling.

3. We show that trace replays of samples generated using BlkSample are more representative of the performance of the full trace compared to samples generated using random spatial sampling.

5.1 Motivation

Tracing in Block Storage Systems. Tracing is used to track different activities in a system, such as system calls, event logs, network traffic, and block I/O requests. In block storage systems, block I/O requests are traced for a variety of purposes, such as workload characterization [92], performance analysis [103], error diagnosis, and threat identification [43]. The contents of a block trace depend on the purpose for which the trace is collected. We use block traces for the performance analysis of block storage systems for which we need the timestamp, the logical block address (LBA), the operation type (read/write) and the size of each block request. Additional features, such as response times, may also be included. Most block storage systems come equipped with tracing tools, and third-party tracing tools have also been developed, but tracing modern block workloads with high request rates consume substantial amounts of storage, network, CPU and memory resources [89]. The resource requirements of tracing can be so severe that system administrators completely avoid it in favor of throwing more hardware to meet the performance requirement by increasing the size of the cache, adding a second tier of cache, or upgrading the storage devices.

Sampling One way of reducing the size of the trace and consequently the resource requirement of workload analysis is to sample block requests instead of tracing every request. Although samples reduce the storage and computation requirements of workload analysis, if samples are not representative of the workload from which it is generated, they can give a wrong idea about the workload and can lead to poor system configuration and design decisions. Random spatial sampling is capable of producing accurate MRCs while sampling as little as 1% of the total workload, significantly reducing the storage and computational overhead of workload analysis [96]. However, block storage system performance, while highly dependent on the miss rate, is dependent on other workload features such as request sizes, IAT and write ratio. Random spatial sampling takes a series of block accesses and generates an accurate MRC. Miss rate alone is insufficient for performance analysis in different scenarios. As we saw in Chapter 4, the reduction in the miss rate does not translate linearly into the increase in bandwidth. Temporal features like IAT influence how the system performance changes due to change in miss rate, so miss rate alone is insufficient in determining whether or not to add a tier-2 cache. So, can we generate block trace samples that do not just translate to accurate MRC but additional features such as read/write request sizes, read/write IATs, read/write misalignment, and write ratio? Such traces would allow us to perform a temporal analysis of the workload, replay traces for better performance analysis, and enable a more long-term analysis of the workload.

Replay Replay is accurate in determining the performance of a given system for a given workload. Replay has been used for a variety of purposes, such as measuring energy efficiency [58], debugging [63], intrusion analysis [27], VM migration [54, 55] and more. Depending on the selectivity and the activity being traced, the resource requirement to collect the trace and replay it can be very high. We replay IO requests from the block to determine the performance of a storage server. IO workloads can be very intensive; hence, traces can be large and can take a long time to replay. Reducing the resource requirement of tracing with minimal sacrifice in accuracy can make tracing a viable approach for many systems.

5.2 Design

We introduce BlkSample, a block trace sampling framework that produces trace samples with accurate workload features such as read/write size, read/write IAT, read/write misalignment, read/write miss ratio and write ratio. The state of the art sampling technique, random spatial sampling, takes a series of block addresses as input and outputs the list of sampled block addresses. However, a block trace has several components such as time, type, and size, and features associated with those components to which random spatial sampling is oblivious. We augment random spatial sampling to output not only a list of sampled block addresses but also a list of block requests. BlkSample outputs a block trace sample with a format identical to and features similar to the sampled request stream. We want to achieve a mean feature error of $\leq 10\%$ while using a sampling rate of $\leq 10\%$. In this section, we begin by describing the shortcomings of our baseline, random spatial sampling, followed by a description of how BlkSample addresses those shortcomings.

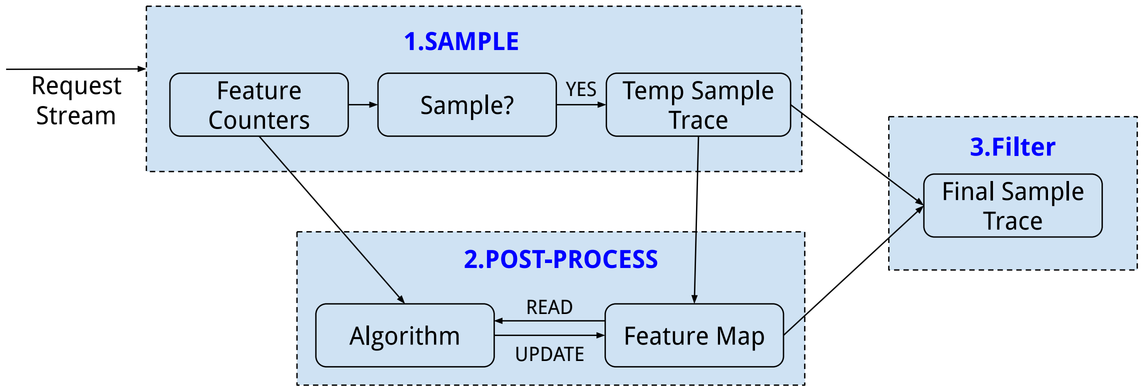


Figure 5.1: The design of BlkSample.

5.2.1 Random Spatial Sampling

Random spatial sampling is used to produce accurate MRCs given a list of block addresses with low sampling rates. It uses a hash value to pick or discard a block address. All accesses to a block address are either sampled or discarded. The addresses used for sampling represent blocks of a fixed size; therefore, a multi-block request has to be broken into individual blocks for sampling. It is oblivious to features such as IAT, size, and type of block requests, which are influential in determining system performance.

The type of block request (read or write) can have a drastically different effect on performance. Writes are slower and reduce the device lifetime of SSDs, whereas reads are cheaper and do not impact device lifetime. The request rate of the workload determines the performance improvement from a cache hit. A cache hit during a period of high request rate is more valuable than a cache hit during a period with relatively low request rate. A workload with large request sizes issues sequential accesses to backing store with different performance compared to a workload with small request sizes, which can issue more random requests to backing store.

One way to generate an identical sample block trace using random spatial sampling is to track the timestamp, request type, and index of the block request for each sampled block. There cannot be a separate block request for each sampled block.

Sampled blocks that are adjacent and belong to the same original block request can be merged to create a new sample block request. Note that a single block request in the request stream can lead to multiple sample block requests because multiple sets of contiguous blocks belonging to the request were sampled. For example, consider a block request that accesses blocks 0 to 10. Using a sampling rate of 50%, we can have 0,1,3,5,6 as sampled blocks. This would lead to 3 sample block requests with blocks (0,1), (3) and (5,6). This can bloat the size of the trace to be stored and distort the features of the sample relative to the original request stream. Next, we discuss the approach we took to avoid such scenarios.

5.2.2 BlkSample

Figure 5.1 illustrates the design of BlkSample. BlkSample generates an accurate block trace sample in 3 steps: sample, post-process, and filter. The sample step creates a temporary trace sample using random spatial sampling with user-specified granularity. The post-process utilizes the feature counters to determine which blocks to remove from the temporary trace sample so that the sample error is reduced. Finally, the filter step will create the final sample trace by removing the block identified in the post-process step from the temporary sample trace generated in the sample step.

Sample The first step of BlkSample is to update the feature counters and sample the request stream. We track workload features using six feature counters: total read/write requests, total read/write size, and total read/write IAT. These counters can be used later to calculate the mean read/write size, the mean read/write IAT, and the write ratio of the workload. Once the feature counter are updated based on the attributes of the block request, the start and end byte offset of the block request is used to generate a list of region addresses accessed by the block request. Each region address is evaluated for sampling, generating a list of sampled regions from the block request. The set of adjacent sampled regions forms a new sample block request. Note that a single block request can have multiple sets of adjacent sample regions leading to multiple sample block requests.

The size of a region is determined by the number of lower-order address of a cache block that is ignored. The start and end byte offsets of a block request maps to a set of cache blocks of size 4KB. When we do not ignore any lower-order address bits, the size of the region being sampled is equal to the size of a cache block which is 4KB. Ignoring 1 lower-order address bit is equal to sampling 8KB regions as now cache blocks 0 and 1 would map to the same region address of 0. Ignoring 2 lower-order address bits is equal to sampling 16KB regions as cache blocks 0, 1, 2 and 3 map to the sample region address of 0. Using larger regions reduces the instances where a single block request splits into multiple sample requests, the phenomenon discussed in Section 5.2.1. Minimizing the split of a single block request into multiple sample block requests will lead to a smaller sample with more representative request sizes.

Post-Process The second phase of post-processing attempts to improve the quality of the temporary sample by further removing accesses to addresses from the sample such that the sample error is reduced. Identifying and removing the accesses to an address that reduces sample error the most is done algorithmically. Filtering all requests to an address and evaluating the resulting sample error would be too expensive, especially to do it for all addresses in the sample. The first step to algorithmically determine the result of removing access to an address in the sample is to compute a hash table of the access characteristics of each address in the sample using Algorithm 2. The hash table maps each address in the temporary sample to a set of 12 features that describe the accesses to the address. The number and total bytes of misaligned read and write requests are 4 out of the 12 features. The rest of the 8 features are the count and total IAT of read and write requests for 4 different types of access: *solo*, *mid*, *left*, and *right*. A *solo* access is when an address is the only address accessed in a block request. A *mid* access is when a lower and higher address is accessed in a block request. Block requests where an address is the most left and the most right in a block request represent *left* and *right* access, respectively. The total space required per block is 48 bytes of data and a 4 byte key. If the working set size of the sample trace is 100GB, we would require approximately 1.3GB of memory to store the features of its addresses representing regions of size 4KB. The memory requirement for storing features can be managed by adjusting the size of region represented by each address. We can reduce the memory required to store the feature from 1.3GB to 0.65GB by using 8KB regions instead of 4KB.

Our post-processing algorithm uses the feature map along with the counter of the full workload to further improve the sample. The algorithm uses the workload counters as a reference to iteratively remove blocks from the sample so that the feature error is reduced. After removing a block, the algorithm makes the necessary adjustments to the feature map. The algorithm terminates once the iteration count is met or if there is no block that can be removed to reduce the feature error. Finally, in the filter phase, all requests in the temporary sample to addresses not in the feature map are filtered out to create the final sample which is smaller but more accurate than the temporary sample.

5.2.3 Algorithm

Ignoring lower-order address bits to increase sampling granularity reduces the mean sample error. But there can be potential to further reduce the error in the sample. During sampling, we used eight counters to track the total read/write byte, the read/write count, the total read/write IAT, and the total read/write misalignment. The features of the full trace, such as the mean read/write request size, the mean read/write IAT, and the write ratio, are derived from these counters. These

```
Result: Hash Table of Access Features
// Inputs:
blksize ← size of a backing store block in bytes ;
cblksize ← size of a cache block in bytes;
sample ← sample block trace;
for each block request in sample do
    size ← size of block request ;
    op ← the read/write operation of block request ;
    iat ← inter-arrival time of block request ;
    lba ← logical block address of block request ;
    startbyte ← lba * blksize;
    endbyte ← startbyte + size;
    startblk ← startbyte//cblksize ;
    endblk ← (endbyte-1)//cblksize ;
    for blk in range(startblk, endblk + 1) do
        frontmisalign ← startbyte%cblksize ;
        endmisalign ← (cblksize − (endbyte%cblksize) ;
        if startblk == endblk then
            table[blk][op].solo + + ;
            table[blk][op].soloiat+ = iat ;
            if (frontmisalign > 0) or (endmisalign > 0) then
                table[blk][op].misalign + + ;
                table[blk][op].misalignbyte+ = frontmisalign ;
                table[blk][op].misalignbyte+ = endmisalign ;
        else if blk == startblk then
            table[blk][op].left + + ;
            table[blk][op].leftiat+ = iat ;
            if frontmisalign > 0 then
                table[blk][op].misalign + + ;
                table[blk][op].misalignbyte+ = frontmisalign ;
        else if blk == endblk then
            table[blk][op].right + + ;
            table[blk][op].rightiat+ = iat ;
            if ENDMISALIGN > 0 then
                table[blk][op].misalign + + ;
                table[blk][op].misalignbyte+ = endmisalign
        else
            table[blk][op].mid + + ;
            table[blk][op].midiat+ = iat ;
return table ;
```

Algorithm 2: Algorithm to compute the features of accesses to a sampled address.

features are used to post-process the sample to reduce the sample error. The algorithm iteratively removes blocks from the sample that reduce the sample error the most.

To remove the block that reduces the sample error the most, we need to evaluate how the workload features would change if the block were to be removed from the sample. A naive approach to evaluate how the workload features of the sample change when a block is removed from the sample is to filter all requests to the block from the sample and recompute the workload features. This approach will be computationally intractable, especially for large samples that can contain millions of requests. We need a faster way to evaluate the change in the features of the sample if a given block was to be removed from the sample. We track the features per block in the sample that enable us to quickly compute new workload features if the block is removed from the sample. These features enable us to algorithmically compute the net change in workload features when a cache block is removed from the block trace. Consequently, we can compute the net change in workload features for all blocks in the sample and remove the block that reduces the mean sample error the most.

Algorithm 2 creates a feature map from a stream of requests in the sample. The features are collected per region, which is set by the user in Algorithm 2 shown on line X. For example, for a block group of size 4KB, the features of 8 512 byte blocks will be aggregated. The size of the group determines the memory and computation requirements of the algorithm. The larger the block group, the lesser the memory and compute requirements, but this comes at the cost of performance. Evaluating and removing smaller groups of blocks will always lead to equal or lower mean sample error compared to evaluating and removing large groups of blocks, but at the cost of more memory and compute resources. A group of size 4KB will have a larger map of features in memory and will require more iterations to determine the best block to remove, but it will be more accurate than using a group size of 16KB. Apart from region size, the other input parameters are the size of a block (also referred to as LBA (Logical Block Address)) in backing store, and the sample block trace. The first step is to loop over each block request in the sample block trace on line 4. We extract the IAT, LBA (Logical Block Address), size, and operation type from the sample block

```
Result: New workload features if all accesses to an address is removed.
// Inputs:
1  rcount ← number of read requests ;
2  wcount ← number of write requests ;
3  rbyte ← total read request bytes ;
4  wbyte ← total write request bytes ;
5  riat ← total read IAT ;
6  wiat ← total write IAT ;
7  rmisalign ← total read misalignment byte ;
8  wmisalign ← total write misalignment byte ;
9  cblksize ← size of a cache block in bytes;
10 features ← features of the block to be removed ;
11 rcount ← rcount − features$[r].solo$ + features$[r].mid$ ;
12 wcount ← wcount − features$[w].solo$ + features$[w].mid$ ;
13 delta_rbyte ← ((features$[r].solo$ + features$[r].mid$ + features$[r].left$ +
     features$[r].right$) ∗ cblksize) − features$[r].misalignbyte$ ;
14 delta_wbyte ← ((features$[w].solo$ + features$[w].mid$ + features$[w].left$ +
     features$[w].right$) ∗ cblksize) − features$[w].misalignbyte$ ;
15 rbyte ← rbyte − delta_rbyte ;
16 wbyte ← wbyte − delta_wbyte ;
17 riat ← riat − features$[r].soloiat$ + features$[r].midiat$) ;
18 wiat ← wiat − features$[w].soloiat$ + features$[w].midiat$) ;
19 rmisalign ← rmisalign − $features[r].misalignbyte$ ;
20 wmisalign ← wmisalign − $features[w].misalignbyte$ ;
21 return rcount, wcount, rbyte, wbyte, riat, wiat, rmisalign, wmisalign
```

Algorithm 3: Compute new workload features if all accesses to a given address is removed.

request. We use the LBA, size of an LBA in bytes, and the size of the region in bytes to compute the start and end block groups accessed by the current block request. Now, we iterate over each region accessed in the block request to track its features. In each iteration, we find whether the block group index is the only (solo) block group being accessed, the left most (first), the right most (last) or the middle block (mid).

Once we have the map of features, we can now iterate over each block group index to find which block group, if removed, reduces the mean sample error the most. We can select the block group index to be removed using different combinations of features or different weights for each feature. This is done until we run out of group block indexes to remove such that the mean sample error is reduced or if we hit a target sampling rate defined by the user.

5.2.4 Replay

We use a custom block trace replay tool that uses the CacheLib cache engine for cache data and libaio to communicate with the backing storage device. To accelerate trace replay, we do not wait for the entire duration of IAT if there are no pending block requests in the system. We use the LRU cache replacement policy in both cache tiers. We compare the replay performance of the full trace with the samples to evaluate if we can infer the performance of the full trace by only replaying the sample.

5.3 Evaluation

In this section, we analyze samples generated with and without using BlkSample to answer the following questions:

- Does random spatial sampling generate block traces that have accurate block features along with accurate hit rates?
- Does sampling groups of blocks instead of individual blocks improve the accu-

```
Result: Features of the given block address removed and the map updated.
table ← hash table storing features per cache block ;
blk ← block to remove ;
if blk − 1 in table then
    table[blk − 1][$r$].$right$+ = table[blk − 1][$r$].$mid$;
    table[blk − 1][$w$].$right$+ = table[blk − 1][$w$].$mid$;
    table[blk − 1][$r$].$mid$, table[blk − 1][$w$].$mid \leftarrow 0, 0$;
    table[blk − 1][$r$].$rightiat$+ = table[blk − 1][$r$].$midiat$;
    table[blk − 1][$w$].$rightiat$+ = table[blk − 1][$w$].$midiat$;
    table[blk − 1][$r$].$midiat$, table[blk − 1][$w$].$midiat \leftarrow 0, 0$;
    table[blk − 1][$r$].$solo$+ = table[blk − 1][$r$].$left$;
    table[blk − 1][$w$].$solo$+ = table[blk − 1][$w$].$left$;
    table[blk − 1][$r$].$left$, table[blk − 1][$w$].$left \leftarrow 0, 0$;
    table[blk − 1][$r$].$soloiat$+ = table[blk − 1][$r$].$leftiat$;
    table[blk − 1][$w$].$soloiat$+ = table[blk − 1][$w$].$leftiat$;
    table[blk − 1][$r$].$leftiat$, table[blk − 1][$w$].$leftiat \leftarrow 0, 0$;
else if blk + 1 in table then
    table[blk + 1][$r$].$left$+ = table[blk + 1][$r$].$mid$;
    table[blk + 1][$w$].$left$+ = table[blk + 1][$w$].$mid$;
    table[blk + 1][$r$].$mid$, table[blk + 1][$w$].$mid \leftarrow 0, 0$;
    table[blk + 1][$r$].$leftiat$+ = table[blk + 1][$r$].$midiat$;
    table[blk + 1][$w$].$leftiat$+ = table[blk + 1][$w$].$midiat$;
    table[blk + 1][$r$].$midiat$, table[blk + 1][$w$].$midiat \leftarrow 0, 0$;
    table[blk + 1][$r$].$solo$+ = table[blk + 1][$R$].$right$;
    table[blk + 1][$w$].$solo$+ = table[blk + 1][$w$].$right$;
    table[blk + 1][$r$].$right$, table[blk + 1][$w$].$right \leftarrow 0, 0$;
    table[blk + 1][$r$].$soloiat$+ = table[blk + 1][$r$].$rightiat$;
    table[blk + 1][$w$].$soloiat$+ = table[blk + 1][$w$].$rightiat$;
    table[blk + 1][$r$].$rightiat$, table[blk + 1][$w$].$rightiat \leftarrow 0, 0$;
$DELETE$ table[blk] ;
```

Algorithm 4: Delete a block from the feature map and make the necessary adjustments.

racy of block features of the sample without hurting the accuracy of hit rates?

- Does the selective removal of blocks from the sample improve the accuracy of block features without significantly hurting the accuracy of hit rates?
- Is there a similarity in the replay performance between the sample and the full block trace?

5.3.1 Random Spatial Sampling

We start by evaluating our baseline, which is random spatial sampling with minor adjustments to track timestamps and request types. Random spatial sampling generates accurate hit rates from a list of addresses that represent blocks of uniform size. However, block requests have variable sizes and can access multiple fixed-size blocks in a single request. A simple fix to avoid this problem is to break up a block request that requests multiple blocks into individual blocks of uniform sizes. However, when multiple sets of adjacent blocks belonging to the same block request are sampled, multiple sample block requests can originate from a single block request. The first row of Figure 5.3 shows the distribution of the request ratios when using different sampling ratios. The request ratio represents the ratio between the request count in the sample and the entire trace. A request ratio greater than 1 means that the number of requests in the sample was greater than in the entire trace. The first row of Figure 5.3 shows that the request ratio not only goes beyond 1 regularly, but can go up to 12. If the request ratio is too high, then it defeats the purpose of sampling, as the storage requirement of the sample will be similar to that of the entire trace.

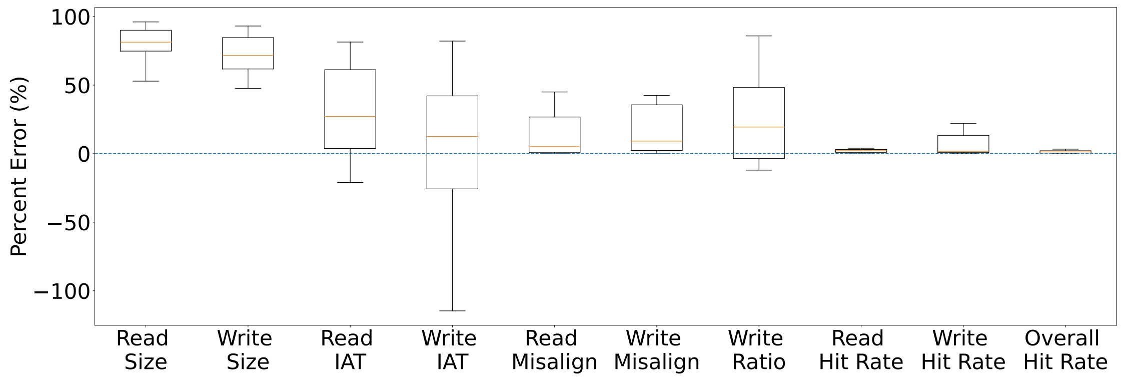
Even if large request ratios mean that samples consume spaces almost equal to the full traces, the reduced working set size could still be helpful in reducing the memory and temporal requirements of workload analysis. We create a sample block request for every set of contiguous blocks. We determine the offset and size of the block request based on the addresses of the blocks in the set. The timestamp and request type of the sample block request are derived from the original block request. We now assess whether block trace samples created using random spatial sampling have accurate workload characteristics such as read/write size, read/write IAT, read/write misalignment, write ratio, and hit rates. An accurate set of features with a smaller working set size could also be useful to accelerate workload analysis.

Figure 5.2 shows the error in the mean value of different features in the samples generated using random spatial sampling with different sampling rates. We see, as expected, that the accuracy of the mean read, write, and overall hit rate is low even when using a sampling rate as low as 5% in Figure 5.2a. On the other hand, we see that features not related to the hit rate, such as size, IAT, and misalignment, have higher error values even when using a sampling rate as high as 80% in Figure 5.2c. The read / write request sizes have the highest error in Figures 5.2a, 5.2b, and 5.2c. Even when using an unreasonably high sampling rate of 80% in Figure 5.2c the minimum error in the size of the read/write request is at least approximately 20%. We see that random spatial sampling, while useful for generating accurate hit rates, is insufficient to generate block trace with accurate read/write size, read/write IAT, read/write misalignment, and write ratio.

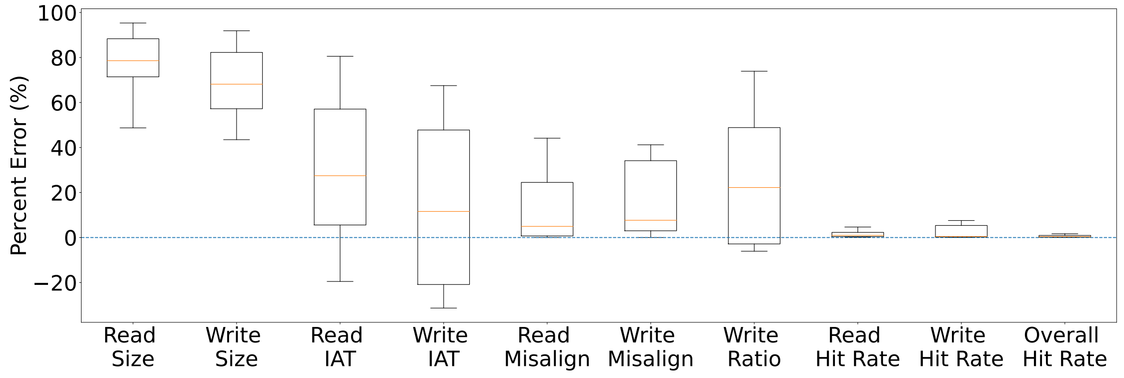
5.3.2 Sampling Granularity

We evaluate the effect of increasing the granularity of random spatial sampling on the accuracy of workload features of the sample block trace. We sample groups of blocks instead of individual blocks by ignoring one or a few lower-order bits of block addresses during sampling. For instance, if we ignore 1 lower-order address bit, blocks 0 and 1 map to the same scaled address of 0. Ignoring 0 bits is our baseline random spatial sampling and ignoring 1, 2 and 4 lower-order bits is using random spatial sampling with increased granularity.

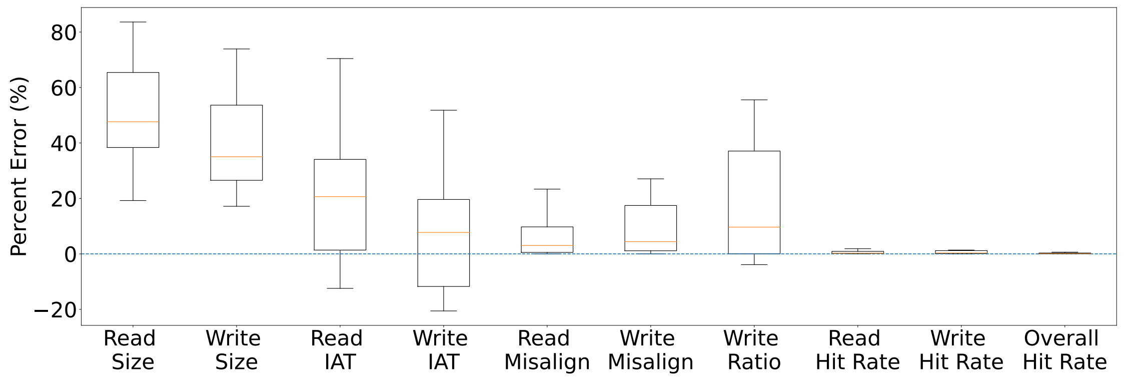
First, we evaluated the sample size of samples with different numbers of lower-



(a) Sampling Rate 5%



(b) Sampling Rate 20%



(c) Sampling Rate 80%

Figure 5.2: Percent error of features in samples generated using random spatial sampling.

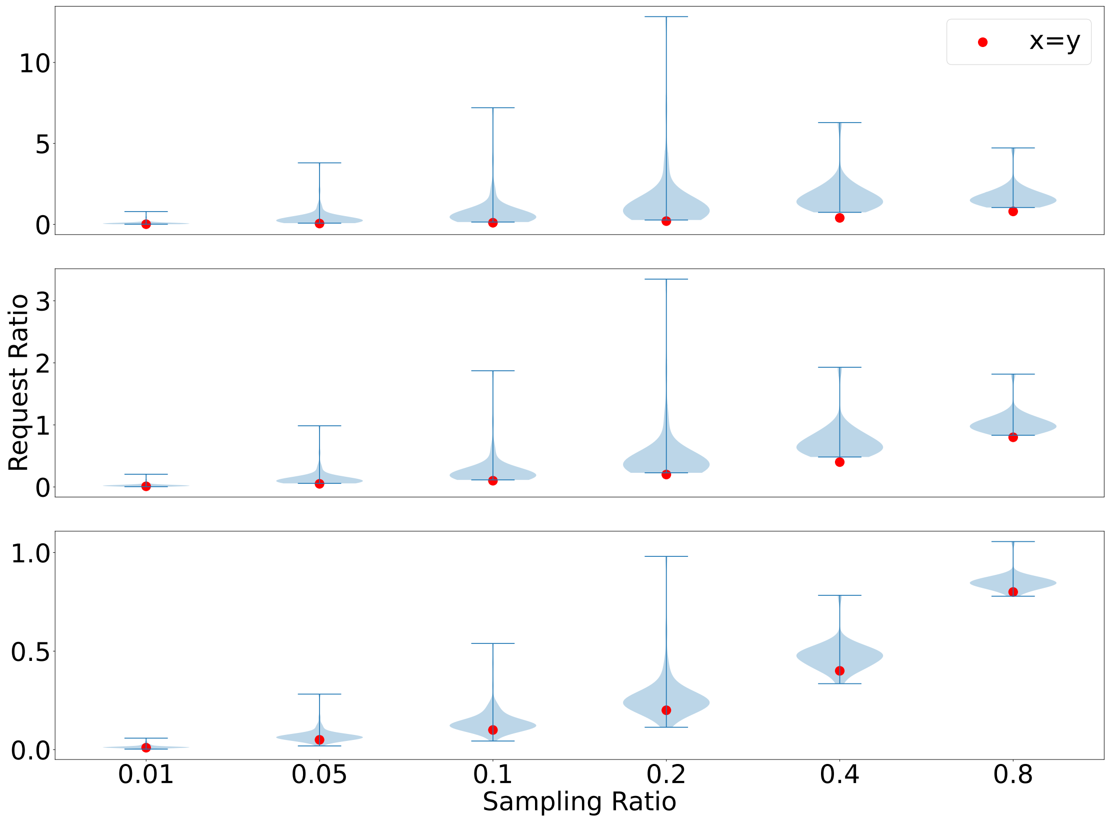


Figure 5.3: Request ratios when ignoring 0 (top), 2 (middle) and 4 (bottom) lower-order address bits under different sampling ratios.

order bits ignored. We have seen the problem of a single block request from the original trace generating multiple sample block requests when we use random spatial sampling. We define the request ratio as the ratio of the number of block requests in the sample to the number of block requests in the full trace. The request ratio of a sample, although not exact, should be close to the sampling ratio. A request ratio close to 1 means that the size of the sample trace is close to that of the full trace. A request ratio of greater than 1 means that the size of the sample is larger than that of the full trace, which defeats the purpose of sampling from a storage space optimization point of view.

Figure 5.3 shows the distribution of the request ratios for samples at different sampling ratios for different numbers of lower-order bits ignored. The red dot represents the point where the request ratio is equal to the sampling ratio. We can see that the size of the y-axis decreases and the red dot moves closer to the dense area of the violin plot as we go from the top (0 bits ignored) to the middle (2 bits ignored) to the bottom (4 bits ignored). Increasing the number of lower-order address bits ignored brought the request ratios closer to the sampling ratios. We can see that for all 3 rows, the length of the violin plot increases to 0.2, after which it starts decreasing. This is because at low sampling rates, the chances that multiple groups of blocks that are not adjacent to each other are sampled are low. At high sampling rates, most of the blocks in the block request might be sampled, so there are less chances that there are multiple groups of sampled blocks, leading to multiple sample block requests. The problem is most prevalent in sampling ratio like 0.2 where it is likely that multiple blocks in a block request are sampled and they are also not adjacent to each other. The first row of Figure 5.3 shows that the request ratio can be as high as 12 when we use a sampling ratio of 0.2. This means that the sample trace had 12x more requests compared to the full trace. The problem is worse in workloads with a large number of blocks accessed per request.

Next, we evaluated the effect of sampling granularity on the features of the sample trace. Figure 5.4 shows the distribution of sample error across different sampling rates while varying sampling granularity by ignoring 0, 1, 2 and 4 lower-order bits of addresses. We see that the error is reduced when we increase the granularity of the sampling. Each sampled block request has its own features such as size and interarrival time, which is better reflected when we use a larger sampling granularity, which reduces the instance of multiple sample requests being generated from a single source block request. Multiple sample requests generated from the same source block requests introduce noise in the overall sample feature, increasing error. We can see that random spatial sampling, which corresponds to 0 lower-order bits ignored, produces a large error value across features even when using a sampling rate as high as

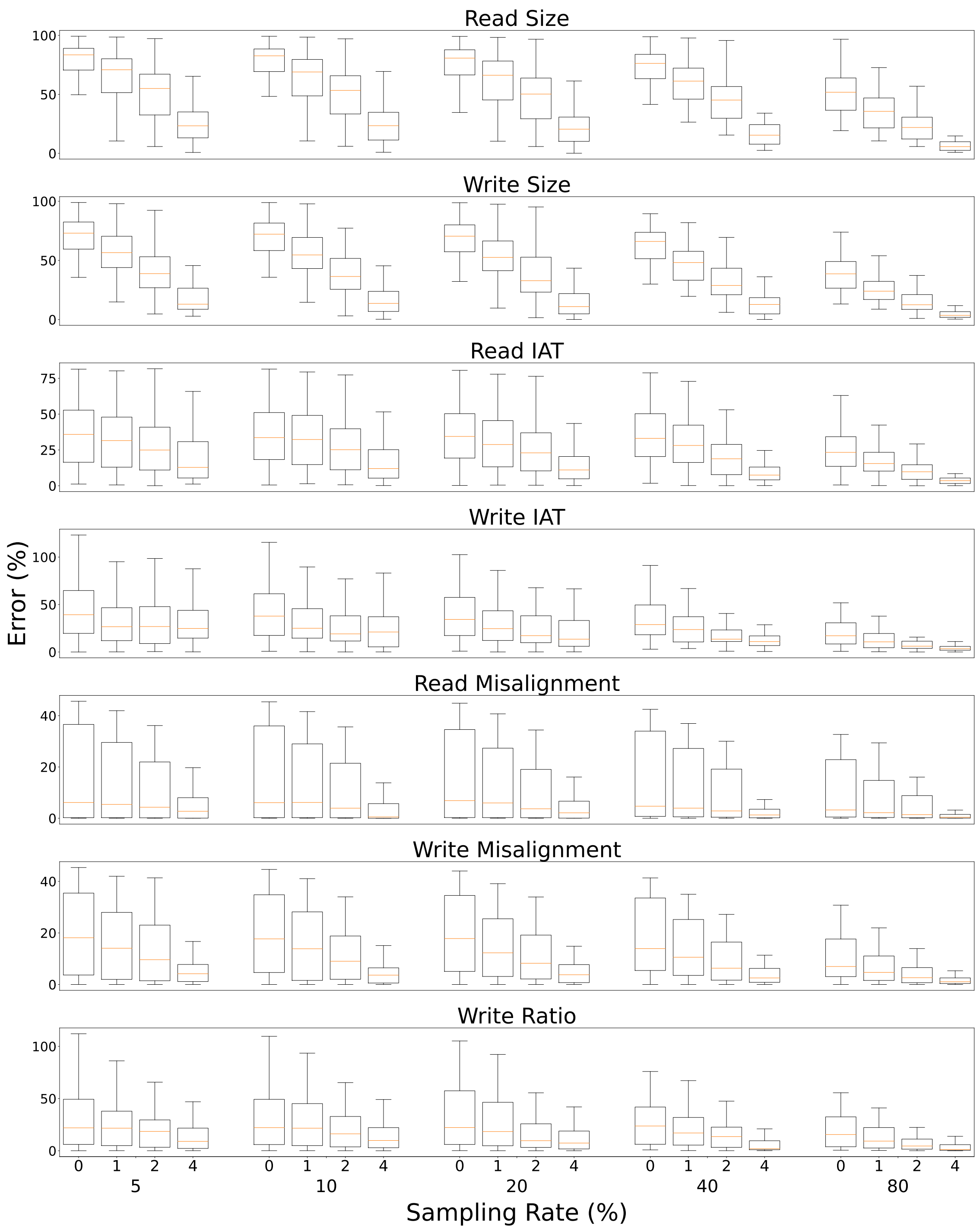


Figure 5.4: A set of 4 box plots representing percent error values when ignoring 0, 1, 2 and 4 lower order address bits at different sampling rates (5%, 10%, 20%, 40%, 80%) for different workload features.

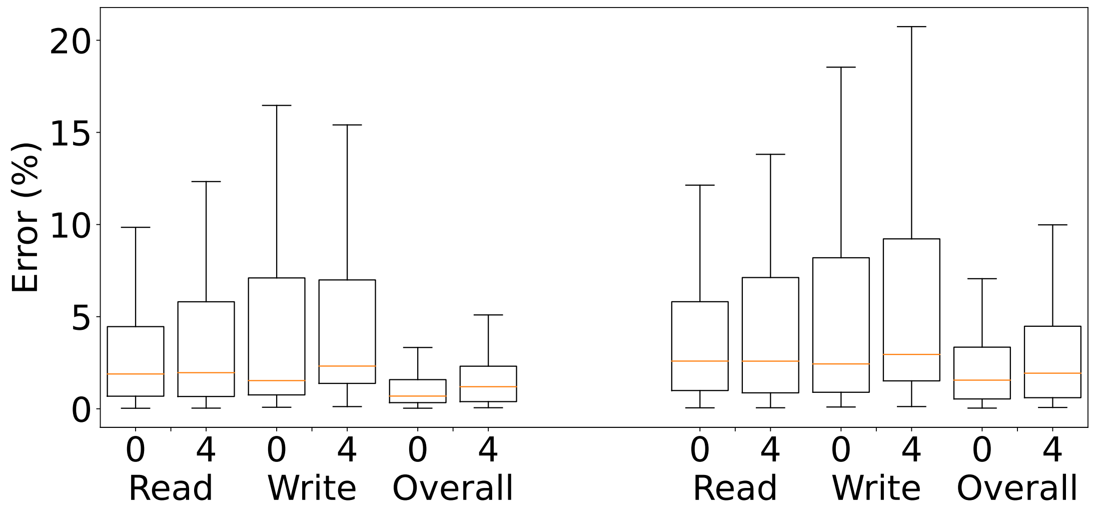


Figure 5.5: Distribution of error in mean (left) and P99 (right) error values in read, write and overall hit rate when ignoring 0 and 4 bits.

80%. This shows that random spatial sampling is not suitable for creating accurate samples. Now, we evaluate whether the improvement in workload features comes at the cost of the accuracy in hit rate that random spatial sampling is very good at. Figure 5.5 shows the error in the hit rate when we use the default random spatial sampling (0 bits ignored) and when we increase the granularity of the sampling (4 bits ignored). We can see that there is minimal increase in the hit rate error when we increase the sampling granularity. The error in the hit rate is magnified when it is divided into read hit rate and write hit rate. Random spatial sampling does not differentiate between a read and a write hit rate. The proven performance of random spatial sampling can be seen in the low error in the overall hit rate. However, when we divide the hit rate between read and write, the error increases but we can see that the sampling granularity does not have a significant error.

5.3.3 Post-processing Algorithm

Increasing the sampling granularity significantly reduces the sample error. We evaluate whether we can further reduce the sample error by removing blocks so that

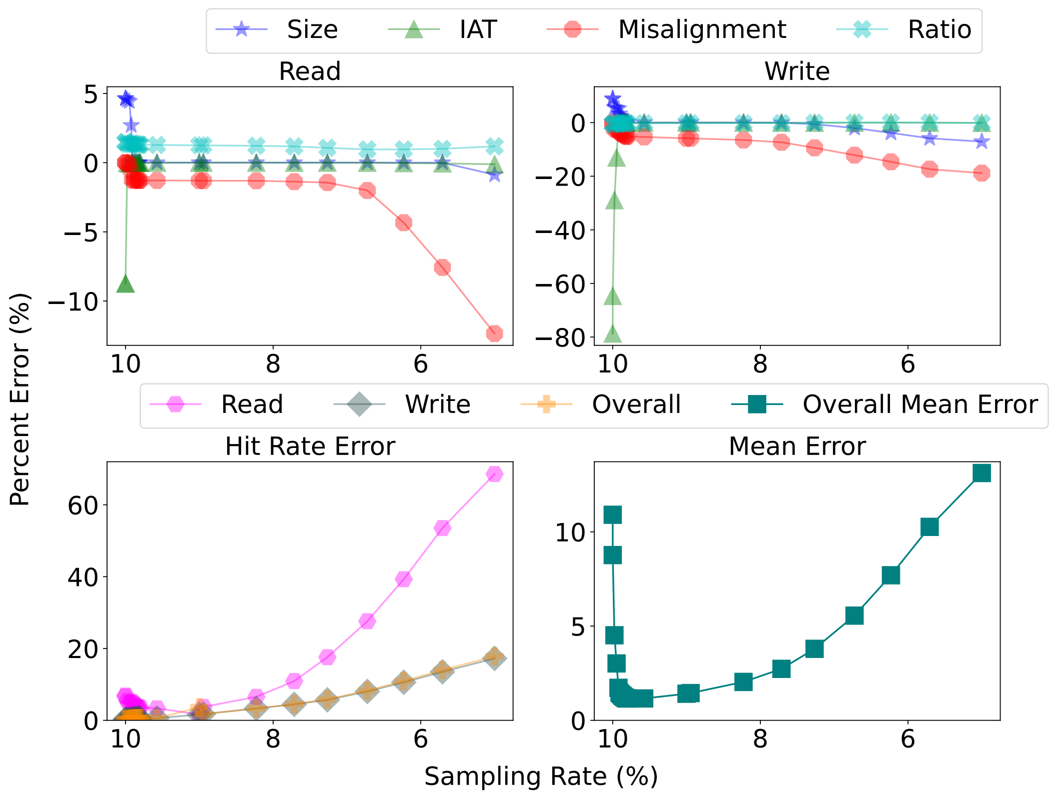


Figure 5.6: Block-trace replay framework design.

the sample error is reduced. We tracked the features of the full workload using counters, which are used as reference when removing blocks. We use Algorithm 2 to generate a mapping of block addresses to a set of access features. We iterate over each block to evaluate the resulting effect of removing the block from the trace using the Algorithm 3. We remove the block that reduces the most sample error and update the feature map using Algorithm 4. We keep doing these 3 steps until we hit a target rate or until we cannot improve the features of the trace anymore.

Figure 5.8 shows the change in percent error values for different features and workloads at each iteration of the post-processing algorithm. We can see that the mean sample error can be reduced by using post-processing. The largest gain occurs in the early iteration, and the gain flattens as we continue to remove blocks. We see that removing blocks brings the most disruption to hit-rate values, which is also

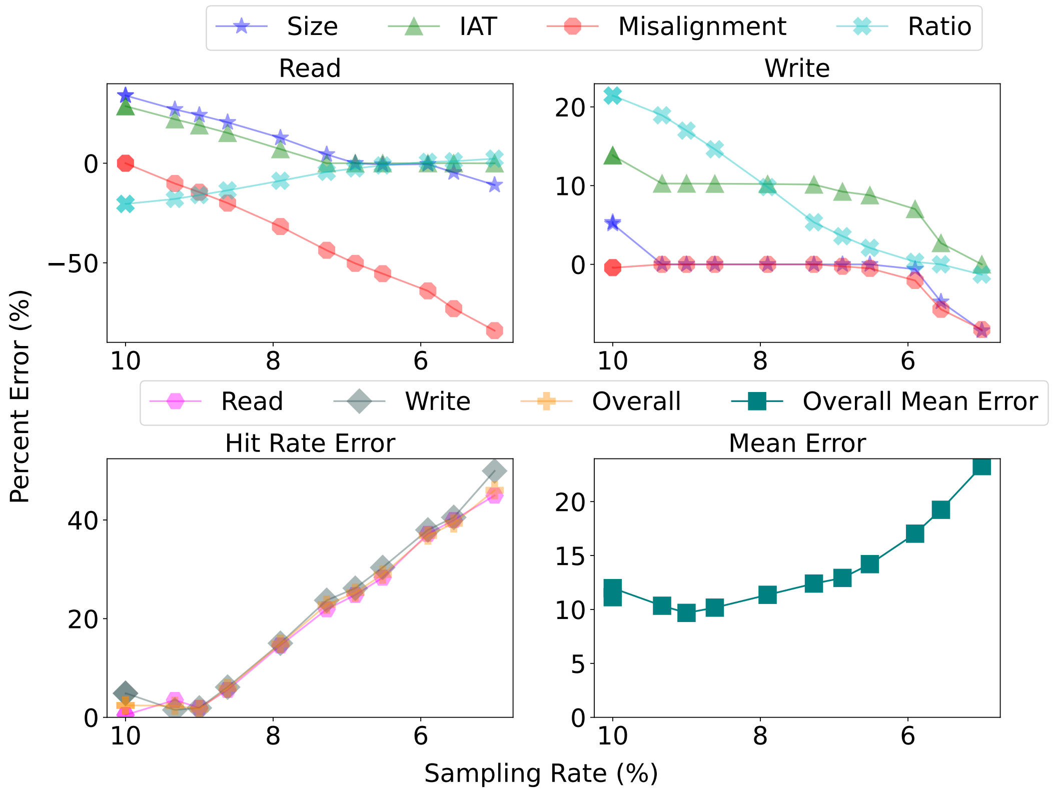


Figure 5.7: Block-trace replay framework design.

indicative that only a few blocks should be removed if there is a large gain in mean error. The post-processing algorithm does not consider the impact of removing a block on the hit rate values because it is very expensive to evaluate. Doing it for all the blocks would take a long time for a single iteration. However, we can see in the last row of Figure 5.8 that even though the post-processing algorithm does not consider hit rate values, the overall mean error is improved even when taking into account the impact on hit rate.

Figures 5.6 and 5.7 show the percentage error in different feature values as we reduce a sample generated with a sampling rate of 10% and ignore four lower-order address bits for workloads w96 and w104, respectively. We can see the U-shaped curve by looking at the mean error graph in the bottom right of both figures. This shows that there is a drop in mean error in the early iterations, and eventually the

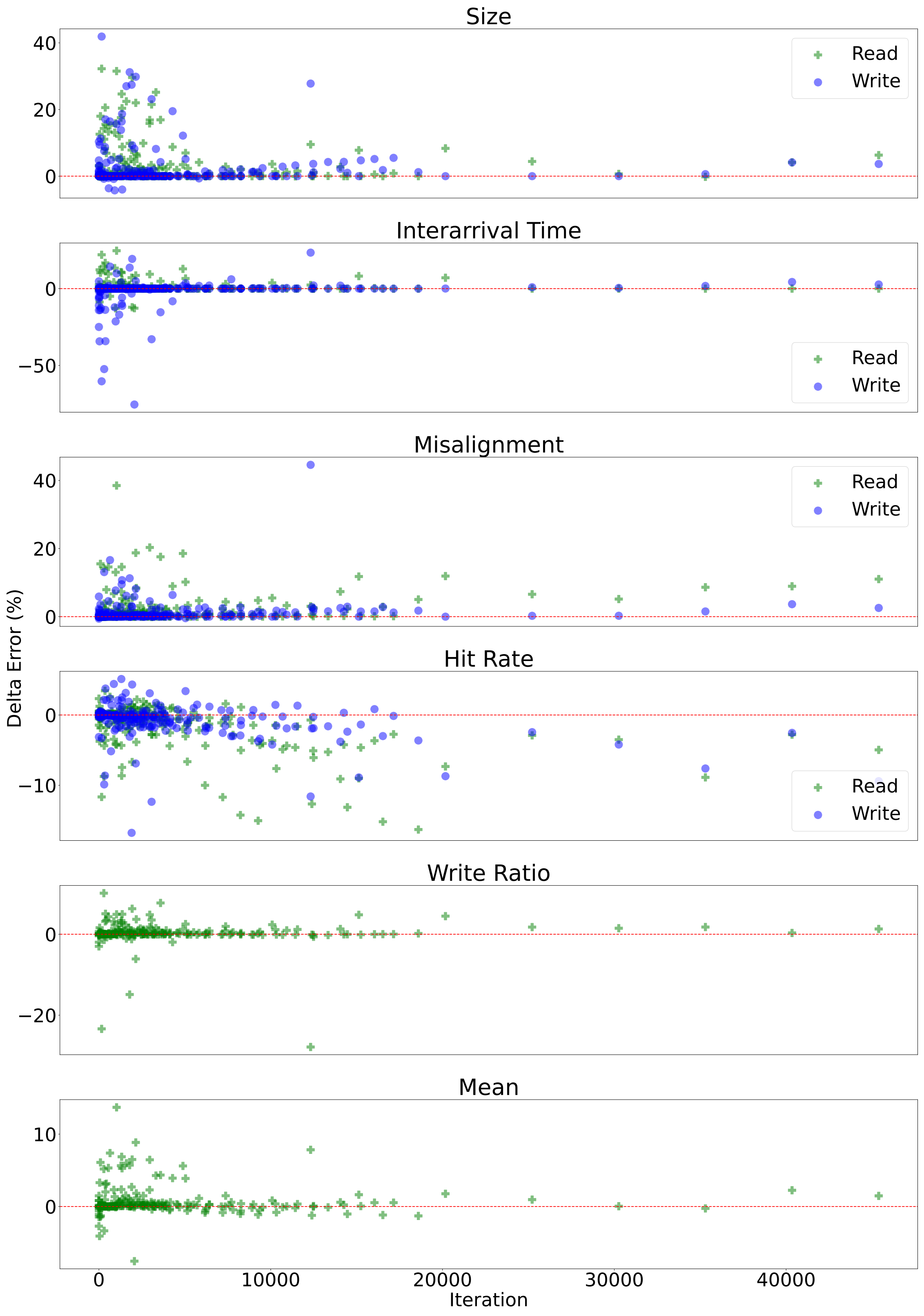


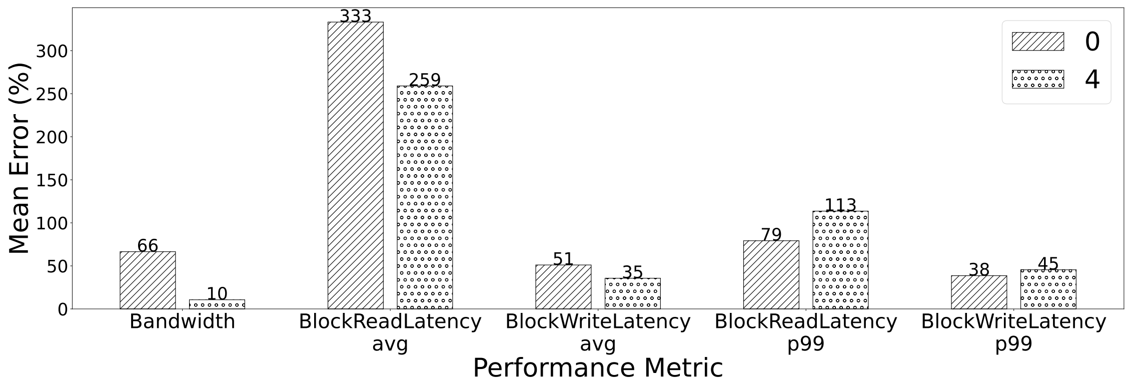
Figure 5.8: The change in sample error after each iteration of the post-processing algorithm across different features. Positive values indicate a reduction in error while negative values indicate an increase.

error value starts increasing. We can see that the improvement in mean error can come from a large reduction in error in one feature, while sacrificing a slight increase in error in other features. We see that the hit rate is the most sensitive to continuous reduction of the sample.

5.3.4 Replay

We replay a full block trace and its sample block traces using different tier-1 and tier-2 cache sizes. Our goal is to evaluate whether the performance metrics of the sample trace replay can be representative of the full trace replay. We chose w96 as the representative workload because the samples are accurate, the working set sizes are large, and the number of requests to be replayed is relatively small. The large working set size is essential so that the samples also have a large enough working set size that meets the minimum tier-1 and tier-2 size requirements of our replay tool. Accurate samples are more likely to perform similarly to the full trace, the large working set size allows us to evaluate a large range of cache sizes with varying hit rates, and the small trace size reduces the time required to replay. Note that the cache size of the sample replay is adjusted on the basis of the sampling rate from which it was generated. For each set of samples we ran 16 trace replays with tier-1 and tier-2 cache sizes scaled based on 25%, 50%, 75% and 100% of the working set size.

We saw in Section 5.3.2 that increasing the sampling granularity reduced the sample error. We evaluated whether the reduced sample error from increasing sampling granularity by ignoring 4 bits translated into more representative replay performance as well. We compare the replay performance of the samples with the sampling rates of 10% and 20% with 0 and 4 lower order bits of block addresses ignored. Figure 5.9 shows the mean error in the performance metrics when we compare the performance of the entire trace with the samples. There is a large discrepancy in the mean and



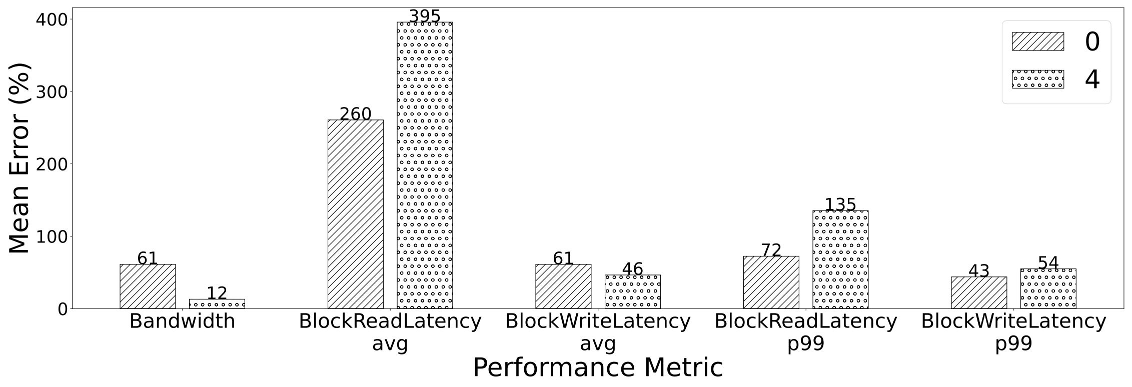


Figure 5.9: The error in replay performance metrics of sample block traces of workload 'w96' across 16 different tier-1 and tier-2 cache size tuples.

p99 latency of the full and sample replays. However, we see that samples generated with large sampling granularity (4 lower-order bits ignored) and sampling rates 10% and 20% have a bandwidth similar to their full trace. This means that we can infer the bandwidth of the full trace replay while only replaying the sample trace with an accuracy of approximately 10% for this workload.

Next, we evaluated additional traces that were generated using large sampling granularity. With sampling granularity set by ignoring 4 lower-order bits of block addresses, we replay samples with sampling rate 10%, 20%, and 40% along with a sample that was generated after post-processing a 10% samples. Figure 5.10 shows the CDF of the percent difference in bandwidth in the replay of 4 samples compared to the replay of the full trace. The tier sizes used for the 16 experiments that we used are identical for all samples. The median bandwidth error for all samples is around

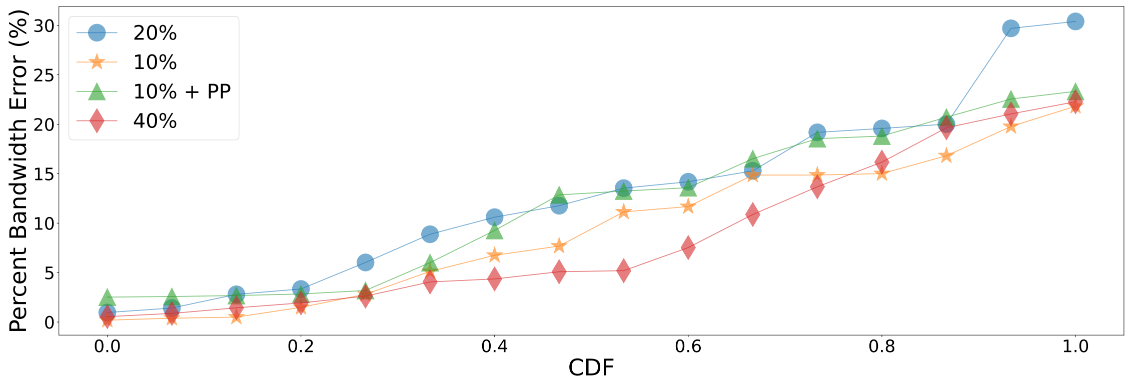


Figure 5.10: CDF of percent error in bandwidth when replaying different samples.

10%. We see that the performance of the sample with the sampling rate 40% is the most accurate, which is what we would expect. However, the sample with a sampling rate 20% performed worse than the sample with 10%. We can also see that post-processing improves feature accuracy of the sample, but that does not necessarily translate to replay performance.

5.4 Discussion

We have presented the design and evaluation of BlkSample. We would like to discuss what we can improve upon, the limitation, and the benefits of sample block traces.

Representative Features We post-processed the samples to bring the mean values of the sample and the full trace as close as possible. It is simpler to determine the chance is mean feature values algorithmically than tracking the quantiles of feature values. However, the quantiles can be highly relevant for specific features such as inter-arrival time and size, and not so much for a feature like misalignment. The spike in request rate can be better replicated if the lower quantiles of interarrival times of the sample and the full trace align. The percentage of workload requests with small sizes can make a big difference in how a workload interacts with storage devices, especially with SSD writes.

Limitation This work focuses on LRU caches. However, modern replacement policies, such as SIEVE, have outperformed LRU [110]. It is important to evaluate how well sampling performs when determining the performance of such modern replacement policies. This work assumes that every output is admitted to the tier-2 cache. This is not an optimal policy for hit rate or performance [109]. Different cache admission policies and their effect on replay performance are an important part that has not yet been explored.

Other Benefits of Sample Block Traces We, especially in academia, are familiar with the situation in which traces of research cannot be released for proprietary reasons. Furthermore, the size of the traces is also a barrier in sharing them. BlkSample can generate multiple traces with representative features and identical format which are smaller in size. Multiple samples generated per workload and identical format can allow researchers to work on workloads without privacy breaches.

5.5 Related Work

The most basic way to model a cache is to run simulations for each cache size. The paper by Mattson introduced an algorithm that uses the concept of inclusion property to generate reuse distances, modeling all cache sizes in a single analysis [60]. Attempts to further improve the space and time requirements for the generation of reuse distances using the inclusion property have included the use of more efficient data structures [19] and parallel processing [64]. Although this drastically reduced the time and resource requirement of cache modeling, the working set size of workloads continued to bloat, and collecting the entirety of the workload became a bottleneck itself. Sampling was used as a possible alternative to generate accurate hit rate estimates for a given workload [96, 41]. Note that given a block trace, random spatial sampling can only generate a list of block accesses. Our technique, on the other hand, generates a sample block trace in an identical format from a source block trace. We introduce sampling regions by ignoring lower-order bits of block addresses and tracking the operation type and IAT of the sampled block request. We do this with minimal compute overhead and without any additional storage required. Trace replay has been a method of analyzing the performance of a server for various workloads[37, 58]. Miniature simulations [95] of workloads has been used to successfully generate hit rates, but we perform the initial evaluation of the performance of a sample when replayed on a server and its comparison to the replay of the full workload.

5.6 Conclusion

The advent of MT caches increases the capacity of a server. The cost and overhead of sampling and analysis increase as the volume of request served by a server increases. MT caching is complex, and efficiently utilizing a tier-2 cache requires the analysis of workload the server serves. This makes sampling a necessity to reduce the size and resource requirement for trace storage and analysis. BlkSample provides a low overhead technique for generating representative trace samples. It builds on random spatial sampling that successfully models the hit rates of the workload. With additional information on the features of the full workload, we are able to adjust the sample to better represent the full workload.

Chapter 6 Conclusion and Future Directions

In this thesis, we investigate how to use MT caches efficiently. We suggest principles for MT cache sizing, train a model to predict the effect of tier sizing, and develop a low-overhead technique to estimate block storage system performance. We summarize what we learned from our attempt to better understand and measure the impact of a second-tier cache on system performance.

MT cache performance depends on synergy between storage devices, tier sizes, and workload. It is not trivial to predict the effect of adding a tier-2 cache on performance. It is important to be able to determine the impact of tier-2 cache on performance, as adding a tier-2 cache can harm performance. The traditional metric, such as hit rate, which is used as a proxy for performance, is adequate to determine whether adding a tier-2 cache would improve performance. A unified hit rate for the tier-1 and tier-2 caches hides how many cache hits occur in each tier. Those tier-1 and tier-2 cache hits have drastically different impacts on performance. Even with separate hit rates for tier-1 and tier-2 caches, we cannot determine whether adding a tier-2 cache would improve throughput. Workload features, such as request rate, size, and write ratio, will influence how performance changes when a tier-2 cache is added. Furthermore, the characteristics of storage devices in block storage systems also determine the tier size and workload characteristics that favor the addition of a tier-2 cache.

Although general guidelines such as tier-2 cache favor workloads with high tier-1 miss rate and low IAT, can help make design choices, we need something more quantifiable. Transparent machine learning models, such as decision trees, can generate clear guidelines that determine whether to add a tier-2 cache. It is complex to derive a model that works on all servers and workloads. However, models per server can be trained by recording the performance of the server by replaying diverse workloads with different tier sizes. The creation of such models requires expensive trace replays, but once trained, the model can quickly determine the effect of adding a tier-2 cache. The main source of overhead is tracing every block request and replaying the collected trace under numerous tier sizes. Sampling can be used to generate trace samples that are more representative of a diverse set of features and share the same trace format. Sampling has been used to efficiently model the hit rates for a given workload but ignores other workload features. A sample block trace with representative request size, inter-arrival time, write ratio, and an identical format drastically reduces the resource requirement of analysis needed to determine how a storage system would perform for a given workload.

6.1 Future Directions

Modeling the effect of MT caches on overall performance is challenging. Although this thesis uncovers the workload and device characteristics that favor MT caching along with tools to estimate the benefit of MT caching, it is only the groundwork required to utilize tier-2 caches efficiently. The following directions grouped by chapter can extend the research presented in the thesis.

Turning the Storage Hierarchy On Its Head: The Strange World of Heterogeneous Tiered Caches We plan to extend our approach to inclusive caching, which keeps duplicate copies in several tiers, as well. Furthermore, we plan to expand our analysis to include modern replacement algorithms such as SIEVE [110]. We want to explore MT caches with a mix of write-back and write-through policies. As part of selecting the right cache size, we would like to use simulation to derive the percentage of successful write-back requests for varying sizes.

Large Scale Study of MT Caching Using Trace Replay The size of the tier-1 and tier-2 caches is just one of many configuration parameters that can be tuned to improve the performance of the tier-2 cache. Since SSDs are used as tier-2 cache, cache engines make adjustments and configurable parameters to manage asymmetric read / write performance and device lifetime [20]. The admission rate, which can be dynamically adjusted based on the request rate, can influence overall performance by avoiding admission to tier-2 cache when it can cause latency spikes in the SSD. Data replacement in the tier-2 cache cannot be done at a block granularity to avoid small writes to SSD, which reduces the device lifetime. Instead, a larger granularity called region size, such as 16MB is used to admit and evict data. Analyzing performance at different region sizes and tier-2 optimized replacement policies can help squeeze more performance from the tier-2 cache.

Modeling per server can be effective in determining how a server will perform with a given workload. The ultimate goal is to develop a universal model that can predict any combination of device characteristics and workload. One of the biggest barriers to developing the model is collecting enough replay data. Large number of replays has to be run across a diverse set of servers, workloads, and configurations to train such models. These models will give us insights into specific device characteristics, workload features, and combination that support or not support a tier-2 cache.

BlkSample: Sampling for Block Storage Traces BlkSample has optimized the trace samples to have representative mean read/write size, read/write IAT, read/write misalignment ratio, and write ratio. We want to expand the features of the sample that we iteratively improve through post-processing. We want to include lower percentiles of size and IAT, which can be as important if not more than the mean values. Furthermore, we want to evaluate our samples on specific storage devices to see if samples can be used for device performance profiling instead of full traces.

Bibliography

[1] Auto Tiering - Redis — redis.io. https://redis.io/auto-tiering/. [Accessed 23-04-2024].

[2] Cache Tiering &x2014; Ceph Documentation — docs.ceph.com. https://docs.ceph.com/en/latest/rados/operations/cache-tiering/. [Accessed 23-04-2024].

[3] Exadata Smart Flash Cache Features and the Oracle Exadata Database Machine. https://www.oracle.com/us/solutions/exadata-smart-flash-cache-366203.pdf. [Accessed 23-04-2024].

[4] Intel® 64 and ia-32 architectures software developer's manual volume 3 (3a, 3b, 3c 3d): System programming guide. 2016.

[5] Industry Perspectives — Nov 12. Don't forget about memory: Dram's surprising role in the high cost of data centers, Nov 2015.

[6] I. Ahmad. Easy and efficient disk I/O workload characterization in VMware ESX server. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2007.

[7] Ibrahim "Umit" Akgun, Ali Selman Aydin, Andrew Burford, Michael McNeill, Michael Arkhangelskiy, and Erez Zadok. Improving storage systems using machine learning. *ACM Transactions on Storage (TOS)*, 1(1):1–30, November 2022.

[8] Ibrahim "Umit" Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. A machine learning framework to improve storage system per-

formance. In *HotStorage '21: Proceedings of the 13th ACM Workshop on Hot Topics in Storage*, Virtual, July 2021. ACM.

[9] Ibrahim "Umit" Akgun, Santiago Vargas, Michael Arkhangelskiy, Andrew Burford, Michael McNeill, Aruna Balasubramanian, Anshul Gandhi, and Erez Zadok. Predicting network buffer capacity for BBR fairness. In *NeurIPS MLSys Workshop*, December 2022.

[10] Ahmad Alwosheel, Sander van Cranenburgh, and Caspar G. Chorus. Is your dataset big enough? sample size requirements when using artificial neural networks for discrete choice analysis. *Journal of Choice Modelling*, 28:167–182, September 2018.

[11] Amazon. A-tech 128gb kit (8x16gb). https://www.amazon.com/Tech-128GB-8x16GB-Memory-PowerEdge/dp/B09YBDRLN3/, January 2023.

[12] Amazon. A-tech 192gb ram. https://www.amazon.com/Tech-ThinkStation-PC4-21300-Registered-288-Pin/dp/B09L5CP767/, January 2023.

[13] Anandtech: Hardware news and tech reviews since 1997, 2022. www.anandtech.com.

[14] Anand Lal Shimpi (Anandtech). Intel ssd dc s3500 review (480gb): Part 1. https://www.anandtech.com/show/7065/intel-ssd-dc-s3500-review-480gb-part-1, June 2013.

[15] Dulcardo Arteaga and Ming Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage*, pages 1–11, 2014.

[16] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the*

12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[17] Jayanta Basak, Kushal Wadhwani, and Kaladhar Voruganti. Storage workload identification. *ACM Transactions on Storage*, 12(3):14:1–14:30, May 2016.

[18] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[19] B. T. Bennett and V. J. Kruskal. Lru stack processing. *IBM J. Res. Dev.*, 19(4):353–357, jul 1975.

[20] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The cachelib caching engine: design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 769–786. USENIX Association, 2020.

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

[22] 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.

[23] 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.

[24] 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.

[25] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Accurate modeling and generation of storage i/o for datacenter workloads. In *Proceedings of the 2nd Workshop on Exascale Evaluation and Research Techniques*, Newport Beach, CA, USA, March 2011. EXERT. https://github.com/Microsoft/diskspd.

[26] Peter Desnoyers. Analytic models of ssd write performance. *ACM Transactions on Storage (TOS)*, 10(2):1–25, 2014.

[27] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through Virtual-Machine logging and replay. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, Boston, MA, December 2002. USENIX Association.

[28] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing

dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Ehsan Elhamifar and René Vidal. Sparse manifold clustering and embedding. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[31] Said Elnaffar, Pat Martin, and Randy Horman. Automatically classifying database workloads. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 622–624. ACM, 2002.

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

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

[34] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[35] 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.

[36] B. Gill. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *FAST*, 2008.

[37] Alireza Haghdoost, Weiping He, Jerry Fredin, and David H.C. Du. On the accuracy and scalability of intensive I/O workload replay. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 315–328, 2017.

[38] 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.

[39] Xubin He, Martha J Kosa, Stephen L Scott, and Christian Engelmann. A unified multiple-level cache for high performance storage systems. *International Journal of High Performance Computing and Networking*, 5(1-2):97–109, 2007.

[40] 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.

[41] 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(2):1–34, 2018.

[42] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 167–181, New York, NY, USA, 2013. ACM.

[43] Hassaan Irshad, Gabriela F. Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Kyu Hyoung Lee, Jignesh M. Patel, Somesh Jha, Yonghwi Kwon, Dongyan Xu, and Xiangyu Zhang. Trace: Enterprise-wide provenance tracking for real-time apt detection. Jan 2021.

[44] Akanksha Jain and Calvin Lin. Hawkeye: Leveraging Belady's algorithm for improved cache replacement. In *2nd Cache Replacement Championship*, Toronto, Ontario, Canada, June 2017.

[45] Eric Burgener John Rydning. High data growth and modern applications drive new storage requirements in digitally transformed enterprises, 2022.

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

[47] Shan Li and H Howie Huang. Black-box performance modeling for solid-state drives. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 391–393. IEEE, 2010.

[48] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, February 2016. USENIX Association.

[49] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-Tier cache management using write hints. In *4th USENIX Conference on File and Storage Technologies (FAST 05)*, San Francisco, CA, December 2005. USENIX Association.

[50] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, page 9, USA, 2005. USENIX Association.

[51] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly tuning tail latencies of cloud systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 981–992, Boston, MA, 2018. USENIX Association.

[52] Kart-Leong Lim, Xudong Jiang, and Chenyu Yi. Deep clustering with variational autoencoder. *IEEE Signal Processing Letters*, 27:231–235, 2020.

[53] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[54] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery.

[55] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery.

[56] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. emrc: Efficient miss ratio approximation for multi-tier caching. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 293–306. USENIX Association, February 2021.

[57] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. emrc: Efficient miss ratio approximation for multi-tier caching. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 293–306. USENIX Association, February 2021.

[58] Zhuo Liu, Fei Wu, Xiao Qin, Changsheng Xie, Jian Zhou, and Jianzong Wang. Tracer: A trace replay tool to evaluate energy-efficiency of mass storage systems. In *2010 IEEE International Conference on Cluster Computing*, pages 68–77. IEEE, 2010.

[59] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 631–645, New York, NY, USA, 2018. ACM.

[60] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, jun 1970.

[61] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*, pages 243–262, 2021.

[62] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vcacheshare: Automated server flash cache space management in a virtualization environment. In *USENIX Annual Technical Conference*, Philadelphia, PA, June 2014.

[63] Robert HB Netzer and Barton P Miller. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[64] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1284–1294, 2012.

[65] Jeong S. Oh, Kyung S. Choi, Jeong R. Kwon, and Sang H. Lee. Finding the near workload type between tpc-c and tpc-w environments. In *International Conference on Convergence and Hybrid Information Technology*, pages 334–337. IEEE, 2008.

[66] Stan Park and Kai Shen. A performance evaluation of scientific i/o workloads on flash-based ssds. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–5. IEEE, 2009.

[67] Dylan Patel and Gerald Wong. Ai server cost analysis – memory is the biggest loser, May 2023.

[68] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan Kaufmann, 2016.

[69] Pankaj Pipada, Achintya Kundu, K. Gopinath, Chiranjib Bhattacharyya, Sai Susarla, and P.C. Nagesh. Loadiq: Learning to identify workload phases from a live storage trace. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage'12, Berkeley, CA, USA, 2012. USENIX Association.

[70] David Plonka, Archit Gupta, and Dale W. Carder. Application buffer-cache management for performance: Running the world's largest mrtg. In *LISA*, 2007.

[71] Lorenzo Posani, Alessio Paccoia, and Marco Moschettini. The carbon footprint of distributed cloud storage. *arXiv preprint arXiv:1803.06973*, 2018.

[72] Anna Povzner, Kimberly Keeton, Arif Merchant, Charles B. Morrey, III, Mustafa Uysal, and Marcos K. Aguilera. Autograph: Automatically extracting workflow file signatures. *SIGOPS Oper. Syst. Rev.*, 43(1):76–83, January 2009.

[73] D. B. Prats, J. L. Berral, and D. Carrera. Automatic generation of workload profiles using unsupervised learning pipelines. *IEEE Transactions on Network and Service Management*, 15(1):142–155, March 2018.

[74] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432, 2006.

[75] 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.

[76] 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.

[77] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with cacheus. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies, FAST 2021*, Proceedings of the 19th USENIX Con-

ference on File and Storage Technologies, FAST 2021, pages 341–354. USENIX Association, 2021.

[78] Stéphane Ross and J. Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *CoRR*, abs/1406.5979, 2014.

[79] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, pages 28:1–28:14, New York, NY, USA, 2014. ACM.

[80] 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.

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

[82] Russell Sears, Catharine Van Ingen, and Jim Gray. To blob or not to blob: Large object storage in a database or a filesystem? *arXiv preprint cs/0701168*, 2007.

[83] Bumjoon Seo, Sooyong Kang, Jongmoo Choi, Jaehyuk Cha, Youjip Won, and Sungroh Yoon. IO workload characterization revisited: A data-mining approach. *IEEE Transactions on Computers*, 63(12):3026–3038, 2014.

[84] Skip Sharipo. Flash Cache Best Practice Guide. https://www.netapp.com/media/19754-tr-3832.pdf. [Accessed 23-04-2024].

[85] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 529–544, USA, 2020. USENIX Association.

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

[87] Rukma Talwadker and Kaladhar Voruganti. Paragone: What's next in block I/O trace modeling. In *Proceedings of the 29th International IEEE Symposium on Mass Storage Systems and Technologies (MSST '13)*, Long Beach, California, May 2013. IEEE.

[88] UserBenchmark, 2022. www.userbenchmark.com.

[89] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. Challenges and solutions for tracing storage systems. Apr 2018.

[90] Michel Verleysen and Damien François. The curse of dimensionality in data mining and time series prediction. In *Computational Intelligence and Bioinspired Systems*, pages 758–770. Springer Berlin Heidelberg, 2005.

[91] 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 Conference on Hot Topics in Storage and File Systems*, HotStorage'18, page 3, USA, 2018. USENIX Association.

[92] K. Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable i/o tracing and analysis. Nov 2009.

[93] VMware. Consolidate applications with less hardware with vsphere hypervisor, Feb 2020.

[94] Muhammad Wajahat, Aditya Yele, Tyler Estro, Anshul Gandhi, and Erez Zadok. Analyzing the distribution fit for storage workload and internet traffic traces. *Performance Evaluation*, pages 102–121, 2020.

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

[96] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.

[97] 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.

[98] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 412–413, New York, NY, USA, 2004. ACM.

[99] 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.

[100] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Baleen: Ml admission & prefetching for flash caches. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies*, FAST '24, USA, 2024. USENIX Association.

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

[102] N. Yadwadkar, C. Bhattacharyya, and K. Gopinath. Discovery of application workloads from network file traces. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '10)*, pages 1–14, San Jose, CA, February 2010. USENIX Association.

[103] S. Yamaguchi, M. Oguchi, and M. Kitsuregawa. Trace system of iscsi storage access. In *The 2005 Symposium on Applications and the Internet*, pages 392–398, 2005.

[104] Juncheng Yang, Ziming Map, Yue Yao, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, Santa Clara, CA, February 2023. USENIX Association.

[105] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 130–149, New York, NY, USA, 2023. Association for Computing Machinery.

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

[107] Zhengyu Yang, Morteza Hoseinzadeh, Ping Wong, John Artoux, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. H-NVMe: A hybrid framework of NVMe-based storage system in cloud computing environment. In *IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2017.

[108] Mohamed Zahran and Sally A. McKee. Global management of cache hierarchies. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, page 131–140, New York, NY, USA, 2010. Association for Computing Machinery.

[109] 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.

[110] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. Sieve is simpler than lru: an efficient turn-key eviction algorithm for web caches.

In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). USENIX Association*, 2024.

[111] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, page 59–72, USA, 2013. USENIX Association.

[112] Yu Zhang, Ping Huang, Ke Zhou, and Hua Wang. Osca: An online-model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (ATC)*, pages 1–13. USENIX Association, 2020.